Quantitative Economics with Julia

JESSE PERLA, THOMAS J. SARGENT AND JOHN STACHURSKI

December 4, 2020
Contents

I  Getting Started with Julia  
1  Setting up Your Julia Environment  3  
2  Interacting with Julia  9  
3  Introductory Examples  17  
4  Julia Essentials  45  
5  Arrays, Tuples, Ranges, and Other Fundamental Types  75  
6  Introduction to Types and Generic Programming  111  

II  Packages and Software Engineering in Julia  135  
7  Generic Programming  137  
8  General Purpose Packages  159  
9  Data and Statistics Packages  167  
10  Solvers, Optimizers, and Automatic Differentiation  177  
11  Julia Tools and Editors  197  
12  Git, GitHub, and Version Control  209  
13  Packages, Testing, and Continuous Integration  227  
14  The Need for Speed  249  

III  Tools and Techniques  271  
15  Linear Algebra  273
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>Orthogonal Projections and Their Applications</td>
<td>297</td>
</tr>
<tr>
<td>17</td>
<td>LLN and CLT</td>
<td>313</td>
</tr>
<tr>
<td>18</td>
<td>Linear State Space Models</td>
<td>331</td>
</tr>
<tr>
<td>19</td>
<td>Finite Markov Chains</td>
<td>355</td>
</tr>
<tr>
<td>20</td>
<td>Continuous State Markov Chains</td>
<td>379</td>
</tr>
<tr>
<td>21</td>
<td>A First Look at the Kalman Filter</td>
<td>401</td>
</tr>
<tr>
<td>22</td>
<td>Numerical Linear Algebra and Factorizations</td>
<td>417</td>
</tr>
<tr>
<td>23</td>
<td>Krylov Methods and Matrix Conditioning</td>
<td>447</td>
</tr>
<tr>
<td>IV</td>
<td>Dynamic Programming</td>
<td>479</td>
</tr>
<tr>
<td>24</td>
<td>Shortest Paths</td>
<td>481</td>
</tr>
<tr>
<td>25</td>
<td>Job Search I: The McCall Search Model</td>
<td>489</td>
</tr>
<tr>
<td>26</td>
<td>Job Search II: Search and Separation</td>
<td>503</td>
</tr>
<tr>
<td>27</td>
<td>A Problem that Stumped Milton Friedman</td>
<td>515</td>
</tr>
<tr>
<td>28</td>
<td>Job Search III: Search with Learning</td>
<td>527</td>
</tr>
<tr>
<td>29</td>
<td>Job Search IV: Modeling Career Choice</td>
<td>543</td>
</tr>
<tr>
<td>30</td>
<td>Job Search V: On-the-Job Search</td>
<td>557</td>
</tr>
<tr>
<td>31</td>
<td>Optimal Growth I: The Stochastic Optimal Growth Model</td>
<td>569</td>
</tr>
<tr>
<td>32</td>
<td>Optimal Growth II: Time Iteration</td>
<td>585</td>
</tr>
<tr>
<td>33</td>
<td>Optimal Growth III: The Endogenous Grid Method</td>
<td>601</td>
</tr>
<tr>
<td>34</td>
<td>LQ Dynamic Programming Problems</td>
<td>609</td>
</tr>
<tr>
<td>35</td>
<td>Optimal Savings I: The Permanent Income Model</td>
<td>637</td>
</tr>
<tr>
<td>36</td>
<td>Optimal Savings II: LQ Techniques</td>
<td>653</td>
</tr>
<tr>
<td>Chapter</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>37</td>
<td>Consumption and Tax Smoothing with Complete and Incomplete Markets</td>
<td>671</td>
</tr>
<tr>
<td>38</td>
<td>Optimal Savings III: Occasionally Binding Constraints</td>
<td>689</td>
</tr>
<tr>
<td>39</td>
<td>Robustness</td>
<td>707</td>
</tr>
<tr>
<td>40</td>
<td>Discrete State Dynamic Programming</td>
<td>727</td>
</tr>
<tr>
<td>V</td>
<td>Modeling in Continuous Time</td>
<td>751</td>
</tr>
<tr>
<td>41</td>
<td>Modeling COVID 19 with Differential Equations</td>
<td>753</td>
</tr>
<tr>
<td>42</td>
<td>Modeling Shocks in COVID 19 with Stochastic Differential Equations</td>
<td>769</td>
</tr>
<tr>
<td>VI</td>
<td>Multiple Agent Models</td>
<td>789</td>
</tr>
<tr>
<td>43</td>
<td>Schelling’s Segregation Model</td>
<td>791</td>
</tr>
<tr>
<td>44</td>
<td>A Lake Model of Employment and Unemployment</td>
<td>801</td>
</tr>
<tr>
<td>45</td>
<td>Rational Expectations Equilibrium</td>
<td>825</td>
</tr>
<tr>
<td>46</td>
<td>Markov Perfect Equilibrium</td>
<td>839</td>
</tr>
<tr>
<td>47</td>
<td>Asset Pricing I: Finite State Models</td>
<td>857</td>
</tr>
<tr>
<td>48</td>
<td>Asset Pricing II: The Lucas Asset Pricing Model</td>
<td>877</td>
</tr>
<tr>
<td>49</td>
<td>Asset Pricing III: Incomplete Markets</td>
<td>887</td>
</tr>
<tr>
<td>50</td>
<td>Uncertainty Traps</td>
<td>899</td>
</tr>
<tr>
<td>51</td>
<td>The Aiyagari Model</td>
<td>911</td>
</tr>
<tr>
<td>52</td>
<td>Default Risk and Income Fluctuations</td>
<td>919</td>
</tr>
<tr>
<td>53</td>
<td>Globalization and Cycles</td>
<td>937</td>
</tr>
<tr>
<td>VII</td>
<td>Time Series Models</td>
<td>951</td>
</tr>
<tr>
<td>54</td>
<td>Covariance Stationary Processes</td>
<td>953</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Pages</td>
</tr>
<tr>
<td>---------</td>
<td>---------------------------------------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>55</td>
<td>Estimation of Spectra</td>
<td>971</td>
</tr>
<tr>
<td>56</td>
<td>Additive Functionals</td>
<td>985</td>
</tr>
<tr>
<td>57</td>
<td>Multiplicative Functionals</td>
<td>1003</td>
</tr>
<tr>
<td>58</td>
<td>Classical Control with Linear Algebra</td>
<td>1023</td>
</tr>
<tr>
<td>59</td>
<td>Classical Filtering With Linear Algebra</td>
<td>1043</td>
</tr>
<tr>
<td>VIII</td>
<td>Dynamic Programming Squared</td>
<td>1063</td>
</tr>
<tr>
<td>60</td>
<td>Dynamic Stackelberg Problems</td>
<td>1065</td>
</tr>
<tr>
<td>61</td>
<td>Optimal Taxation in an LQ Economy</td>
<td>1091</td>
</tr>
<tr>
<td>62</td>
<td>Optimal Taxation with State-Contingent Debt</td>
<td>1111</td>
</tr>
<tr>
<td>63</td>
<td>Optimal Taxation without State-Contingent Debt</td>
<td>1143</td>
</tr>
</tbody>
</table>
Part I

Getting Started with Julia
Chapter 1

Setting up Your Julia Environment

1.1 Contents

- Overview 1.2
- A Note on Jupyter 1.3
- Desktop Installation of Julia and Jupyter 1.4
- Using Julia on the Web 1.5
- Installing Packages 1.6

1.2 Overview

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

There are a few different options for using Julia, including a local desktop installation and Jupyter hosted on the web.

If you have access to a web-based Jupyter and Julia setup, it is typically the most straightforward way to get started.

1.3 A Note on Jupyter

Like Python and R, and unlike products such as Matlab and Stata, there is a looser connection between Julia as a programming language and Julia as a specific development environment.

While you will eventually use other editors, there are some advantages to starting with the Jupyter environment while learning Julia.

- The ability to mix formatted text (including mathematical expressions) and code in a single document.
- Nicely formatted output including tables, figures, animation, video, etc.
- Conversion tools to generate PDF slides, static HTML, etc.

Online Jupyter may be available, and requires no installation.

We’ll discuss the workflow on these features in the next lecture.
1.4 Desktop Installation of Julia and Jupyter

If you want to install these tools locally on your machine

- Download and install Julia, from download page, accepting all default options.
  - We do not recommend JuliaPro.

- Open Julia, by either
  1. Navigating to Julia through your menus or desktop icons (Windows, Mac), or
  2. Opening a terminal and typing `julia` (Linux; to set this up on Mac, see end of section)

You should now be looking at something like this

![Julia REPL](image)

This is called the JULIA REPL (Read-Evaluate-Print-Loop), which we discuss more later.

- In the Julia REPL, hit ] to enter package mode and then enter.

```
add IJulia InstantiateFromURL
```

This adds packages for

- The IJulia kernel which links Julia to Jupyter (i.e., allows your browser to run Julia code, manage Julia packages, etc.).
- The InstantiateFromURL which is a tool written by the QE team to manage package dependencies for the lectures.

Note: To set up the Julia terminal command on Mac, open a terminal and run `sudo ln -s <where_julia_app_is>/Contents/Resources/julia/bin/julia /usr/local/bin/julia`

The full command might look like `sudo ln -s /Applications/Julia-1.4.app/Contents/Resources/julia/bin/julia /usr/local/bin/julia`, if you placed the app in your Applications folder.

**Note:** To obtain the full set of packages we use, at this stage you can run the following (see the package setup section.)

```
using InstantiateFromURL
github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0", instantiate = true, precompile = true)
```

1.4.1 Installing Jupyter

If you have previously installed Jupyter (e.g., installing Anaconda Python by downloading the binary [https://www.anaconda.com/download/](https://www.anaconda.com/download/)) then the add IJulia installs everything you need into your existing environment.
1.4. DESKTOP INSTALLATION OF JULIA AND JUPYTER

Otherwise - or in addition - you can install it directly from the Julia REPL

using IJulia; jupyterlab()

Choose the default, y if asked to install Jupyter and then JupyterLab via Conda.
After the installation, a JupyterLab tab should open in your browser.
(Optional) To enable launching JupyterLab from a terminal, use add Julia’s Jupyter to your path.

1.4.2 Starting Jupyter

Next, let’s install the QuantEcon lecture notes to our machine and run them (for more details on the tools we’ll use, see our lecture on version control).

1. Install git.

2. (Optional, but strongly recommended) Install the GitHub Desktop.

GitHub Desktop Approach

After installing the Git Desktop application, click this link on your desktop computer to automatically install the notebooks.
It should open a window in the GitHub desktop app like this

Choose a path you like and clone the repo.

Note: the workflow will be easiest if you clone the repo to the default location relative to the home folder for your user.

From a Julia REPL, start JupyterLab by executing
using IJulia; jupyterlab()

Alternatively, if you installed Jupyter separately in Jupyter Installation or added Jupyter to your path then run `jupyter lab` in your terminal.

Navigate to the location you stored the lecture notes, and open the Interacting with Julia notebook to explore this interface and start writing code.

**Git Command Line Approach**

If you do not wish to install the GitHub Desktop, you can get the notebooks using the Git command-line tool.

Open a new terminal session and run

`git clone https://github.com/quantecon/quantecon-notebooks-julia`

This will download the repository with the notebooks in the working directory.

Then, `cd` to that location in your Mac, Linux, or Windows PowerShell terminal

`cd quantecon-notebooks-julia`

Then, either using the `using IJulia; jupyterlab()` or execute `jupyter lab` within your shell.

And open the Interacting With Julia lecture (the file `julia_environment.ipynb` in the list of notebooks in JupyterLab) to continue.

### 1.5 Using Julia on the Web

If you have access to an online Julia installation, it is the easiest way to get started.

Eventually, you will want to do a local installation in order to use other tools and editors such as Atom/Juno, but don’t let the environment get in the way of learning the language.

#### 1.5.1 Using Julia with JupyterHub

If you have access to a web-based solution for Jupyter, then that is typically a straightforward option

- Students: ask your department if these resources are available.
- Universities and workgroups: email `[mailto:contact@quantecon.org”]{contact@quantecon.org}` for help on setting up a shared JupyterHub instance with precompiled packages ready for these lecture notes.

**Obtaining Notebooks**

Your first step is to get a copy of the notebooks in your JupyterHub environment.
While you can individually download the notebooks from the website, the easiest way to access the notebooks is usually to clone the repository with Git into your JupyterHub environment.

JupyterHub installations have different methods for cloning repositories, with which you can use the url for the notebooks repository: https://github.com/QuantEcon/quantecon-notebooks-julia.

1.6 Installing Packages

After you have some of the notebooks available, as in above, these lectures depend on functionality (like packages for plotting, benchmarking, and statistics) that are not installed with every Jupyter installation on the web.

If your online Jupyter does not come with QuantEcon packages pre-installed, you can install the InstantiateFromURL package, which is a tool written by the QE team to manage package dependencies for the lectures.

To add this package, in an online Jupyter notebook run (typically with <Shift-Enter>)

```
In [1]: ] add InstantiateFromURL
```

Then, run

```
using InstantiateFromURL
github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0", instantiate = true, precompile = true)
```

If your online Jupyter environment does not have the packages pre-installed, it may take 15-20 minutes for your first QuantEcon notebook to run.

After this step, open the downloaded Interacting with Julia notebook to begin writing code.

If the QuantEcon notebooks do not work after this installation step, you may need to speak to the JupyterHub administrator.
Chapter 2

Interacting with Julia

2.1 Contents

- Overview 2.2
- Using Jupyter 2.3
- Using the REPL 2.4
- (Optional) Adding Jupyter to the Path 2.5

2.2 Overview

In this lecture we’ll start examining different features of the Julia and Jupyter environments.

2.3 Using Jupyter

2.3.1 Getting Started

Recall that the easiest way to get started with these notebooks is to follow the cloning instructions earlier.

To summarize, if on a desktop you should clone the notebooks repository https://github.com/quantecon/quantecon-notebooks-julia, then in a Julia REPL type

using IJulia; jupyterlab()

Hint: Julia will remember the last commands in the REPL, so you can use up-arrow to restart JupyterLab.

Alternatively, if you are using an online Jupyter, then you can directly open a new notebook.

Finally, if you installed Jupyter separately or have added added Jupyter to the Path then cd to the folder location in a terminal, and run

jupyter lab

Regardless, your web browser should open to a page that looks something like this
The page you are looking at is called the “dashboard”. If you click on “Julia 1.x.x” 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.

### 2.3.2 Notebook Basics

Notice that in the previous figure the cell is surrounded by a blue border. This means that the cell is selected, and double-clicking will place it 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`
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.

1. Command mode
   - The green border is replaced by a blue border.
   - 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 it’s 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

Note that if you’re using a JupyterHub setup, you will need to first run

```julia
In [1]: using InstantiateFromURL
   # optionally add arguments to force installation: instantiate = true,
   # precompile = true
   github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")
```

in a new cell (i.e., Shift + Enter).

This might take 15-20 minutes depending on your setup, as it installs a large set of packages for our use.

Run the following cell

```julia
In [2]: using Plots
   gr(fmt=:png);
   plot(sin, -2π, 2π, label="sin(x)")
```

You’ll see something like this (although the style of plot depends on your installation)
2.3. USING JUPYTER

Note: The “time-to-first-plot” in Julia takes a while, since it needs to compile many functions - but is almost instantaneous the second time you run the cell.

2.3.3 Working with the Notebook

Let’s go over some more Jupyter notebook features — enough so that we can press ahead with programming.

Tab Completion

Tab completion in Jupyter makes it easy to find Julia commands and functions available. For example if you type `rep` and hit the tab key you’ll get a list of all commands that start with `rep`.

Getting Help

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

\[ \exp(i \pi) = -1 \]

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

\[ \exp(i \pi) = -1 \]

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 Jupyter 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.
2.4. USING THE REPL

Package Operations

You can execute package operations in the notebook by prepending a `]`. For example, `] st` will give the status of installed packages in the current environment.

Note: Cells where you use `;` and `]` must not have any other instructions in them (i.e., they should be one-liners).

2.3.4 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.

The Jupyter organization has a site for sharing notebooks called nbviewer which provides a static HTML representations of notebooks.

QuantEcon also hosts the QuantEcon Notes website, where you can upload and share your notebooks with other economists and the QuantEcon community.

2.4 Using the REPL

As we saw in the desktop installation, the REPL is a Julia specific terminal. It becomes increasingly important as you learn Julia, and you will find it to be a useful tool for interacting with Julia and installing packages.

As a reminder, to open the REPL on your desktop, either

1. Navigating to Julia through your menus or desktop icons (Windows, Mac), or
2. Opening a terminal and typing `julia` (Linux)

If you are using a JupyterHub installation, you can start the REPL in JupyterLab by choosing

1. Choose “New Launcher”
2. Choose a Julia Console

We examine the REPL and its different modes in more detail in the tools and editors lecture.

2.5 (Optional) Adding Jupyter to the Path

If you installed Jupyter using Julia, then you may find it convenient to add it to your system path in order to launch JupyterLab without running a Julia terminal.
The default location for the Jupyter binaries is relative to the `.julia` folder (e.g., "C:\Users\USERNAME\.julia\conda\3\Scripts on Windows).

You can find the directory in a Julia REPL using by executing

```julia
] add Conda
using Conda
Conda.SCRIPTDIR
```

On Linux/OSX, you could add that path to your `.bashrc`.

On Windows, to add directly to the path, type `;` to enter shell mode and then execute

```
setx PATH "$(Conda.SCRIPTDIR);%PATH%"
```
Chapter 3

Introductory Examples

3.1 Contents

- Overview 3.2
- Example: Plotting a White Noise Process 3.3
- Example: Variations on Fixed Points 3.4
- Exercises 3.5
- Solutions 3.6

3.2 Overview

We’re now ready to start learning the Julia language itself.

3.2.1 Level

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

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

- variables
- arrays or vectors
- loops
- conditionals (if/else)

3.2.2 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

In this lecture, we will often start with a direct MATLAB/FORTRAN approach which often is poor coding style in Julia, but then move towards more elegant code which is tightly
connected to the mathematics.

### 3.2.3 Set Up

We assume that you’ve worked your way through our getting started lecture already.

In particular, the easiest way to install and precompile all the Julia packages used in QuantEcon notes is to type `] add InstantiateFromURL` and then work in a Jupyter notebook, as described here.

### 3.2.4 Other References

The definitive reference is Julia’s own documentation.

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

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

### 3.3 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.

#### 3.3.1 Introduction to Packages

The first step is to activate a project environment, which is encapsulated by `Project.toml` and `Manifest.toml` files.

There are three ways to install packages and versions (where the first two methods are discouraged, since they may lead to package versions out-of-sync with the notes)

1. `add` the packages directly into your global installation (e.g. `Pkg.add("MyPackage"")` or `] add MyPackage`)

2. download an `Project.toml` and `Manifest.toml` file in the same directory as the notebook (i.e. from the `@__DIR__` argument), and then call using `Pkg; Pkg.activate(@__DIR__);`

3. use the `InstantiateFromURL` package

   ```
   In [1]: using InstantiateFromURL
   \# optionally add arguments to force installation: instantiate = true, precompile = true
   github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")
   ```

If you have never run this code on a particular computer, it is likely to take a long time as it downloads, installs, and compiles all dependent packages.

This code will download and install project files from the lecture repo.
We will discuss it more in Tools and Editors, but these files provide a listing of packages and versions used by the code.

This ensures that an environment for running code is reproducible, so that anyone can replicate the precise set of package and versions used in construction.

The careful selection of package versions is crucial for reproducibility, as otherwise your code can be broken by changes to packages out of your control.

After the installation and activation, using provides a way to say that a particular code or notebook will use the package.

In [2]: using LinearAlgebra, Statistics

3.3.2 Using Functions from a Package

Some functions are built into the base Julia, such as randn, which returns a single draw from a normal distribution with mean 0 and variance 1 if given no parameters.

In [3]: randn()
Out[3]: 0.1340729773546092

Other functions require importing all of the names from an external library

In [4]: using Plots
   gr(fmt=:png); # setting for easier display in jupyter notebooks
   n = 100
   ϵ = randn(n)
   plot(1:n, ϵ)

Out[4]:

![Plot of a white noise process](image)
Let’s break this down and see how it works.

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

Because we used Pkg.activate previously, it will use whatever version of Plots.jl that was specified in the Project.toml and Manifest.toml files.

The other packages LinearAlgebra and Statistics are base Julia libraries, but require an explicit using.

The arguments to plot are the numbers 1,2, ..., n for the x-axis, a vector ϵ for the y-axis, and (optional) settings.

The function randn(n) returns a column vector n random draws from a normal distribution with mean 0 and variance 1.

### 3.3.3 Arrays

As a language intended for mathematical and scientific computing, Julia has strong support for using unicode characters.

In the above case, the ϵ and many other symbols can be typed in most Julia editor by providing the LaTeX and <TAB>, i.e. \epsilon<TAB>.

The return type is one of the most fundamental Julia data types: an array

```
In [5]: typeof(ϵ)
```

```
Out[5]: Array{Float64,1}
```

```
In [6]: ϵ[1:5]
```

```
Out[6]: 5-element Array{Float64,1}:
    0.3508933329262338
  -1.3417023907846528
    0.3098019819945068
  -2.4886691920076305
    1.3432653930816776
```

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

In Julia, one-dimensional arrays are interpreted as column vectors for purposes of linear algebra.

The ϵ[1:5] returns an array of the first 5 elements of ϵ.

Notice from the above that

- array indices start at 1 (like MATLAB and Fortran, but unlike Python and C)
- array elements are referenced using square brackets (unlike MATLAB and Fortran)

To get help and examples in Jupyter or other julia editor, use the ? before a function name or syntax.
3.3. EXAMPLE: PLOTTING A WHITE NOISE PROCESS

3.3. EXAMPLE: PLOTTING A WHITE NOISE PROCESS

?typeof

search: typeof typejoin TypeError

Get the concrete type of x.

Examples

julia> a = 1//2;

julia> typeof(a)
Rational{Int64}

julia> M = [1 2; 3.5 4];

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

3.3.4 For Loops

Although there’s no need in terms of what we wanted to achieve with our program, for the sake of learning syntax let’s rewrite our program to use a for loop for generating the data.

Note

In Julia v0.7 and up, the rules for variables accessed in for and while loops can be sensitive to how they are used (and variables can sometimes require a global as part of the declaration). We strongly advise you to avoid top level (i.e. in the REPL or outside of functions) for and while loops outside of Jupyter notebooks. This issue does not apply when used within functions.

Starting with the most direct version, and pretending we are in a world where randn can only return a single value

In [7]: # poor style
n = 100
ϵ = zeros(n)
for i in 1:n
    ϵ[i] = randn()
end

Here we first declared ϵ to be a vector of n numbers, initialized by the floating point 0.0.
The for loop then populates this array by successive calls to randn().
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 either ∈ or =.
The index variable is looped over for all integers from 1:n – but this does not actually create a vector of those indices.
Instead, it creates an iterator that is looped over – in this case the range of integers from 1 to n.

While this example successfully fills in $\epsilon$ with the correct values, it is very indirect as the connection between the index $i$ and the $\epsilon$ vector is unclear.

To fix this, use eachindex

In [8]: # better style
n = 100
$\epsilon$ = zeros(n)
for i in eachindex($\epsilon$)
    $\epsilon[i]$ = randn()
end

Here, eachindex($\epsilon$) returns an iterator of indices which can be used to access $\epsilon$.

While iterators are memory efficient because the elements are generated on the fly rather than stored in memory, the main benefit is (1) it can lead to code which is clearer and less prone to typos; and (2) it allows the compiler flexibility to creatively generate fast code.

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

In [9]: $\epsilon$ _sum = 0.0 # careful to use 0.0 here, instead of 0
m = 5
for $\epsilon$ _val in $\epsilon$[1:m]
    $\epsilon$ _sum = $\epsilon$ _sum + $\epsilon$ _val
end
$\epsilon$ _mean = $\epsilon$ _sum / m

Out[9]: 0.12255423860142103

where $\epsilon[1:m]$ returns the elements of the vector at indices 1 to m.

Of course, in Julia there are built in functions to perform this calculation which we can compare against

In [10]: $\epsilon$ _mean = mean($\epsilon$[1:m])
$\epsilon$ _mean = sum($\epsilon$[1:m]) / m

Out[10]: true

In these examples, note the use of $\approx$ to test equality, rather than $\approx=$, which is appropriate for integers and other types.

Approximately equal, typed with \approx<TAB>, is the appropriate way to compare any floating point numbers due to the standard issues of floating point math.

### 3.3.5 User-Defined Functions

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

To make things more interesting, instead of directly plotting the draws from the distribution, let’s plot the squares of these draws
3.3. **EXAMPLE: PLOTTING A WHITE NOISE PROCESS**

In [11]: # poor style
    function generatedata(n)
        ε = zeros(n)
        for i in eachindex(ε)
            ε[i] = (randn())^2 # squaring the result
        end
        return ε
    end

data = generatedata(10)
plot(data)

Out[11]:

Here

- **function** is a Julia keyword that indicates the start of a function definition
- **generatedata** is an arbitrary name for the function
- **return** is a keyword indicating the return value, as is often unnecessary

Let us make this example slightly better by “remembering” that **randn** can return a vectors.

In [12]: # still poor style
    function generatedata(n)
        ε = randn(n) # use built in function
        for i in eachindex(ε)
            ε[i] = ε[i]^2 # squaring the result
        end
        return ε
    end

data = generatedata(5)
While better, the looping over the $i$ index to square the results is difficult to read. Instead of looping, we can **broadcast** the $^2$ square function over a vector using a ..

To be clear, unlike Python, R, and MATLAB (to a lesser extent), the reason to drop the `for` is **not** for performance reasons, but rather because of code clarity.

Loops of this sort are at least as efficient as vectorized approach in compiled languages like Julia, so use a for loop if you think it makes the code more clear.

In [13]: # better style
   
   function generatedata(n)
       ϵ = randn(n) # use built in function
       return ϵ.^2
   end
   
data = generatedata(5)

Out[13]: 5-element Array{Float64,1}:
   1.4581291740342703
   1.8125501043209953
   0.24904149884873786
   0.04206678337831965
   3.0583819640808283

We can even drop the `function` if we define it on a single line.

In [14]: # good style
  
  generatedata(n) = randn(n).^2
  data = generatedata(5)

Out[14]: 5-element Array{Float64,1}:
   0.23661834702911036
   0.35774378676197877
   4.730500187602342
   0.05446363165615865
   0.7890340301810831

Finally, we can broadcast any function, where squaring is only a special case.

In [15]: # good style
   
   f(x) = x^2 # simple square function
   generatedata(n) = f.(randn(n)) # uses broadcast for some function `f`
   data = generatedata(5)

Out[15]: 5-element Array{Float64,1}:
   0.26883451157964544
   0.8296323684434822
   0.022319579094377283
   0.1644791401573571
   0.02347767867665137
As a final – abstract – approach, we can make the `generatedata` function able to generically apply to a function.

```julia
In [16]: generatedata(n, gen) = gen.(randn(n))  # uses broadcast for some function

f(x) = x^2  # simple square function
data = generatedata(5, f)  # applies f

Out[16]: 5-element Array{Float64,1}:
0.10155073167168607
2.2552140007754518
0.7007155569314104
6.311468975188948
0.11904096398760988
```

Whether this example is better or worse than the previous version depends on how it is used. High degrees of abstraction and generality, e.g. passing in a function \( f \) in this case, can make code either clearer or more confusing, but Julia enables you to use these techniques **with no performance overhead**.

For this particular case, the clearest and most general solution is probably the simplest.

```julia
In [17]: # direct solution with broadcasting, and small user-defined function
    n = 100
    f(x) = x^2
    x = randn(n)
    plot(f.(x), label="x^2")
    plot!(x, label="x")  # layer on the same plot

Out[17]:
```

![Plot](image.png)
While broadcasting above superficially looks like vectorizing functions in MATLAB, or Python ufuncs, it is much richer and built on core foundations of the language.

The other additional function plot! adds a graph to the existing plot.

This follows a general convention in Julia, where a function that modifies the arguments or a global state has a ! at the end of its name.

### A Slightly More Useful Function

Let’s make a slightly more useful function.

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

In doing so we’ll make use of the Distributions package, which we assume was instantiated above with the project.

Here’s the code

In [18]: using Distributions

```julia
function plothistogram(distribution, n)
    ϵ = rand(distribution, n) # n draws from distribution
    histogram(ϵ)

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

Out[18]:

![Histogram of observations](image.png)
Let’s have a casual discussion of how all this works while leaving technical details for later in the lectures.

First, \( lp = \text{Laplace()} \) creates an instance of a data type defined in the \texttt{Distributions} module that represents the Laplace distribution. The name \( lp \) is bound to this value.

When we make the function call \texttt{plothistogram(lp, 500)} the code in the body of the function \texttt{plothistogram} is run with

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

A Mystery

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

This looks like something of a mystery.

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

In [19]: \texttt{rand(3)}

Out[19]: 3-element Array{Float64,1}:
0.43077285177861757
0.07691366276079359
0.08457588208380429

On the other hand, \texttt{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 \texttt{rand()} is able to take this kind of value as an argument and return the output that we want?

The answer in a nutshell is \texttt{multiple dispatch}, which Julia uses to implement \texttt{generic programming}.

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 value.

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

In Julia these alternative versions of a function are called \texttt{methods}.

3.4 Example: Variations on Fixed Points

Take a mapping \( f : X \to X \) for some set \( X \).

If there exists an \( x^* \in X \) such that \( f(x^*) = x^* \), then \( x^* \) is called a “fixed point” of \( f \).
For our second example, we will start with a simple example of determining fixed points of a function.

The goal is to start with code in a MATLAB style, and move towards a more Julian style with high mathematical clarity.

### 3.4.1 Fixed Point Maps

Consider the simple equation, where the scalars $p, \beta$ are given, and $v$ is the scalar we wish to solve for

$$v = p + \beta v$$

Of course, in this simple example, with parameter restrictions this can be solved as $v = p/(1 - \beta)$.

Rearrange the equation in terms of a map $f(x) : \mathbb{R} \rightarrow \mathbb{R}$

$$v = f(v)$$

where

$$f(v) := p + \beta v$$

Therefore, a fixed point $v^*$ of $f(\cdot)$ is a solution to the above problem.

### 3.4.2 While Loops

One approach to finding a fixed point of (1) is to start with an initial value, and iterate the map

$$v^{n+1} = f(v^n)$$

For this exact $f$ function, we can see the convergence to $v = p/(1 - \beta)$ when $|\beta| < 1$ by iterating backwards and taking $n \rightarrow \infty$

$$v^{n+1} = p + \beta v^n = p + \beta p + \beta^2 v^{n-1} = p \sum_{i=0}^{n-1} \beta^i + \beta^n v_0$$

To implement the iteration in (2), we start by solving this problem with a while loop.

The syntax for the while loop contains no surprises, and looks nearly identical to a MATLAB implementation.

In [20]: # poor style
    p = 1.0 # note 1.0 rather than 1
    β = 0.9
    maxiter = 1000
    tolerance = 1.0E-7
    v_iv = 0.8 # initial condition
### 3.4. EXAMPLE: VARIATIONS ON FIXED POINTS

```plaintext
# setup the algorithm
v_old = v_iv
normdiff = Inf
iter = 1
while normdiff > tolerance && iter <= maxiter
    v_new = p + β * v_old # the f(v) map
    normdiff = norm(v_new - v_old)

    # replace and continue
    v_old = v_new
    iter = iter + 1
end
println("Fixed point = $v_old, and \|f(x) - x\| = $normdiff in $iter\) iterations")

Fixed point = 9.999999173706609, and \|f(x) - x\| = 9.181037796679448e-8 in 155 iterations
```

The `while` loop, like the `for` loop should only be used directly in Jupyter or the inside of a function.

Here, we have used the `norm` function (from the `LinearAlgebra` base library) to compare the values.

The other new function is the `println` with the string interpolation, which splices the value of an expression or variable prefixed by $ into a string.

An alternative approach is to use a `for` loop, and check for convergence in each iteration.

```plaintext
In [21]: # setup the algorithm
v_old = v_iv
normdiff = Inf
iter = 1
for i in 1:maxiter
    v_new = p + β * v_old # the f(v) map
    normdiff = norm(v_new - v_old)
    if normdiff < tolerance # check convergence
        iter = i
        break # converged, exit loop
    end
# replace and continue
v_old = v_new
println("Fixed point = $v_old, and \|f(x) - x\| = $normdiff in $iter\) iterations")

Fixed point = 9.999999081896231, and \|f(x) - x\| = 9.181037796679448e-8 in 154 iterations
```

The new feature there is `break`, which leaves a `for` or `while` loop.
3.4.3 Using a Function

The first problem with this setup is that it depends on being sequentially run – which can be easily remedied with a function.

In [22]: # better, but still poor style
    
    function v_fp(β, ρ, v_iv, tolerance, maxiter)
        # setup the algorithm
        v_old = v_iv
        normdiff = Inf
        iter = 1
        while normdiff > tolerance && iter <= maxiter
            v_new = p + β * v_old # the f(v) map
            normdiff = norm(v_new - v_old)
            # replace and continue
            v_old = v_new
            iter = iter + 1
        end
        return (v_old, normdiff, iter) # returns a tuple
    end

    # some values
    p = 1.0 # note 1.0 rather than 1
    β = 0.9
    maxiter = 1000
    tolerance = 1.0E-7
    v_initial = 0.8 # initial condition

    v_star, normdiff, iter = v_fp(β, p, v_initial, tolerance, maxiter)
    println("Fixed point = $v_star, and |f(x) - x| = $normdiff in $iter iterations")

    Fixed point = 9.999999173706609, and |f(x) - x| = 9.181037796679448e-8 in 155 iterations

While better, there could still be improvements.

3.4.4 Passing a Function

The chief issue is that the algorithm (finding a fixed point) is reusable and generic, while the function we calculate p + β * v is specific to our problem.

A key feature of languages like Julia, is the ability to efficiently handle functions passed to other functions.

In [23]: # better style
    
    function fixedpointmap(f, iv, tolerance, maxiter)
        # setup the algorithm
        x_old = iv
        normdiff = Inf
        iter = 1
        while normdiff > tolerance && iter <= maxiter
            x_new = f(x_old) # use the passed in map
            normdiff = norm(x_new - x_old)
            x_old = x_new
            iter = iter + 1
        end
        return (x_old, normdiff, iter)
    end

    # some values
    p = 1.0 # note 1.0 rather than 1
    β = 0.9
    maxiter = 1000
    tolerance = 1.0E-7
    v_initial = 0.8 # initial condition

    f(v) = p + β * v

    v_star, normdiff, iter = fixedpointmap(f, v_initial, tolerance, maxiter)
    println("Fixed point = $v_star, and |f(x) - x| = $normdiff in $iter iterations")

    Fixed point = 9.999999173706609, and |f(x) - x| = 9.181037796679448e-8 in 155 iterations
3.4. EXAMPLE: VARIATIONS ON FIXED POINTS

```plaintext
normdiff = norm(x_new - x_old)
x_old = x_new
iter = iter + 1
end
return (x_old, normdiff, iter)
end

# define a map and parameters
p = 1.0
β = 0.9
f(v) = p + β * v  # note that p and β are used in the function!

maxiter = 1000
tolerance = 1.0E-7
v_initial = 0.8  # initial condition

v_star, normdiff, iter = fixedpointmap(f, v_initial, tolerance, maxiter)
println("Fixed point = $v_star, and |f(x) - x| = $normdiff in $iter iterations")

Fixed point = 9.999999173706609, and |f(x) - x| = 9.181037796679448e-8 in 155 iterations

Much closer, but there are still hidden bugs if the user orders the settings or returns types wrong.

3.4.5 Named Arguments and Return Values

To enable this, Julia has two features: named function parameters, and named tuples

```
Fixed point = 9.999999918629035, and |f(x) - x| = 9.041219328764782e-9 in 177 iterations

In this example, all function parameters after the ; in the list, must be called by name. Furthermore, a default value may be enabled – so the named parameter iv is required while tolerance and maxiter have default values.

The return type of the function also has named fields, value, normdiff, and iter – all accessed intuitively using ..

To show the flexibility of this code, we can use it to find a fixed point of the non-linear logistic equation, \( x = f(x) \) where \( f(x) := rx(1 - x) \).

In [25]: r = 2.0
   f(x) = r * x * (1 - x)

   sol = fixedpointmap(f, iv=0.8)
   println("Fixed point = $(sol.value), and |f(x) - x| = $(sol.normdiff) in 
   
   $(sol.iter) iterations")

   Fixed point = 0.4999999999999968, and |f(x) - x| = 3.979330237546819e-8 in 7 iterations

3.4.6 Using a Package

But best of all is to avoid writing code altogether.

In [26]: # best style
   using NLSolve

   p = 1.0
   β = 0.9
   f(v) = p .+ β * v # broadcast the +
   sol = fixedpoint(f, [0.8])
   println("Fixed point = $(sol.zero), and |f(x) - x| = $(norm(f(sol.zero) - 
   
   "* 
   "$(sol.iterations) iterations")

   Fixed point = [9.999999999999973], and |f(x) - x| = 3.552713678800501e-15 in 3 iterations

The fixedpoint function from the NLSolve.jl library implements the simple fixed point iteration scheme above.

Since the NLSolve library only accepts vector based inputs, we needed to make the \( f(v) \) function broadcast on the + sign, and pass in the initial condition as a vector of length 1 with \([0.8] \).

While a key benefit of using a package is that the code is clearer, and the implementation is tested, by using an orthogonal library we also enable performance improvements.
In [27]: # best style
  p = 1.0
  β = 0.9
  iv = [0.8]
  sol = fixedpoint(v -> p .+ β * v, iv)
  println("Fixed point = \$\{(sol.zero), and \ |f(x) - x| = \$\{(norm(f(sol.zero) - sol.zero)) in
     
    
    
    "\$(sol.iterations) iterations")

Fixed point = [9.999999999999973], and \ |f(x) - x| = 3.552713678800501e-15 in 3 iterations

Note that this completes in 3 iterations vs 177 for the naive fixed point iteration algorithm. Since Anderson iteration is doing more calculations in an iteration, whether it is faster or not would depend on the complexity of the f function.

But this demonstrates the value of keeping the math separate from the algorithm, since by decoupling the mathematical definition of the fixed point from the implementation in (2), we were able to exploit new algorithms for finding a fixed point.

The only other change in this function is the move from directly defining \(f(v)\) and using an anonymous function.

Similar to anonymous functions in MATLAB, and lambda functions in Python, Julia enables the creation of small functions without any names.

The code \(v \rightarrow p .+ β * v\) defines a function of a dummy argument, \(v\) with the same body as our \(f(x)\).

### 3.4.7 Composing Packages

A key benefit of using Julia is that you can compose various packages, types, and techniques, without making changes to your underlying source.

As an example, consider if we want to solve the model with a higher-precision, as floating points cannot be distinguished beyond the machine epsilon for that type (recall that computers approximate real numbers to the nearest binary of a given precision; the machine epsilon is the smallest nonzero magnitude).

In Julia, this number can be calculated as

In [28]: eps()

Out[28]: 2.220446049250313e-16

For many cases, this is sufficient precision – but consider that in iterative algorithms applied millions of times, those small differences can add up.

The only change we will need to our model in order to use a different floating point type is to call the function with an arbitrary precision floating point, \(\text{BigFloat}\), for the initial value.

In [29]: # use arbitrary precision floating points
  p = 1.0
\[
\beta = 0.9
\]
\[
iv = [\text{BigFloat}(0.8)] \quad \# \text{higher precision}
\]

# otherwise identical
\[
sol = \text{fixedpoint}(v \to p + \beta \cdot v, iv)
\]
\[
\text{println}("Fixed point = $(sol.zero), and } |f(x) - x| = $(\text{norm}(f(sol.zero) - sol.zero)) \) in
\]

"$(sol.iterations) iterations"

Fixed point = BigFloat[10.
-0000000000000002220446049250313573885329099314128483772878678
09936811555686155], and |f(x) - x| = 0.0 in 3 iterations

Here, the literal `BigFloat(0.8)` takes the number `0.8` and changes it to an arbitrary precision number.

The result is that the residual is now exactly `0.0` since it is able to use arbitrary precision in the calculations, and the solution has a finite-precision solution with those parameters.

### 3.4.8 Multivariate Fixed Point Maps

The above example can be extended to multivariate maps without any modifications to the fixed point iteration code.

Using our own, homegrown iteration and simply passing in a bivariate map:

In [30]:
```
p = [1.0, 2.0]
β = 0.9
iv = [0.8, 2.0]
f(v) = p + β * v  # note that p and β are used in the function!
```

\[
sol = \text{fixedpointmap}(f, iv = iv, tolerance = 1.0E-8)
\]
\[
\text{println}("Fixed point = $(sol.value), and } |f(x) - x| = $(sol.normdiff) \) in
\]

"$(sol.iter) iterations"

Fixed point = [9.9999999961080519, 19.999999923053192], and |f(x) - x| =
9.5e-9 in 184 iterations

This also works without any modifications with the `fixedpoint` library function.

In [31]:
```
using NLsolve
```

\[
p = [1.0, 2.0, 0.1]
β = 0.9
iv = [0.8, 2.0, 51.0]
f(v) = p + β * v
```

\[
sol = \text{fixedpoint}(v \to p + \beta \cdot v, iv)
\]
\[
\text{println}("Fixed point = $(sol.zero), and } |f(x) - x| = $(\text{norm}(f(sol.zero) - sol.zero)) \) in
\]

"$(sol.iterations) iterations"
Fixed point = [10.0, 20.000000000000004, 0.9999999999999929], and |f(x) - x| = 6.661338147750939e-16 in 3 iterations

Finally, to demonstrate the importance of composing different libraries, use a StaticArrays.jl type, which provides an efficient implementation for small arrays and matrices.

In [32]: using NLsolve, StaticArrays
   p = @SVector [1.0, 2.0, 0.1]
   β = 0.9
   iv = [0.8, 2.0, 51.0]
   f(v) = p .+ β * v

   sol = fixedpoint(v -> p .+ β * v, iv)
   println("Fixed point = $(sol.zero), and |f(x) - x| = $(norm(f(sol.zero) - sol.zero)) in 
   " *
   "$(sol.iterations) iterations")

Fixed point = [10.0, 20.000000000000004, 0.9999999999999929], and |f(x) - x| = 6.661338147750939e-16 in 3 iterations

The @SVector in front of the [1.0, 2.0, 0.1] is a macro for turning a vector literal into a static vector.

All macros in Julia are prefixed by @ in the name, and manipulate the code prior to compilation.

We will see a variety of macros, and discuss the “metaprogramming” behind them in a later lecture.

3.5 Exercises

3.5.1 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 \texttt{factorial(n)}.

Write your own version of this function, called \texttt{factorial2}, using a \texttt{for} loop.

3.5.2 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 \texttt{rand()} from the set of Julia’s built-in random number generators (not the Distributions package), write a function \texttt{binomial_rv} such that \texttt{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 $p$.

### 3.5.3 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 larger, the fraction that falls in $B$ converges to the probability of landing in $B$.
- For a circle, area = $\pi \times \text{radius}^2$.

### 3.5.4 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.

### 3.5.5 Exercise 5

Simulate and plot the correlated time series

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

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

Set $n = 200$ and $\alpha = 0.9$.

### 3.5.6 Exercise 6

Plot three simulated time series, one for each of the cases $\alpha = 0$, $\alpha = 0.8$ and $\alpha = 0.98$.

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

### 3.5.7 Exercise 7

This exercise is more challenging.

Take a random walk, starting from $x_0 = 1$

$$x_{t+1} = \alpha x_t + \sigma \epsilon_{t+1} \quad \text{where} \quad x_0 = 1 \quad \text{and} \quad t = 0, \ldots, t_{\text{max}}$$
3.5. **EXERCISES**

- Furthermore, assume that the $x_{t_{\text{max}}} = 0$ (i.e. at $t_{\text{max}}$, the value drops to zero, regardless of its current state).
- The sequence of shocks $\{\epsilon_t\}$ is assumed to be iid and standard normal.
- For a given path $\{x_t\}$ define a **first-passage time** as $T_a = \min\{t \mid x_t \leq a\}$, where by the assumption of the process $T_a \leq t_{\text{max}}$.

Start with $\sigma = 0.2$, $\alpha = 1.0$

1. calculate the first-passage time, $T_0$, for 100 simulated random walks – to a $t_{\text{max}} = 200$ and plot a histogram

2. plot the sample mean of $T_0$ from the simulation for $\alpha \in \{0.8, 1.0, 1.2\}$

### 3.5.8 **Exercise 8(a)**

This exercise is more challenging.

The root of a univariate function $f(\cdot)$ is an $x$ such that $f(x) = 0$.

One solution method to find local roots of smooth functions is called Newton’s method.

Starting with an $x_0$ guess, a function $f(\cdot)$ and the first-derivative $f'(\cdot)$, the algorithm is to repeat

$$x^{n+1} = x^n - \frac{f(x^n)}{f'(x^n)}$$

until $|x^{n+1} - x^n|$ is below a tolerance

1. Use a variation of the **fixedpointmap** code to implement Newton’s method, where the function would accept arguments $f$, $f_{\text{prime}}$, $x_0$, $\text{tolerance}$, $\text{maxiter}$.

2. Test it with $f(x) = (x - 1)^3$ and another function of your choice where you can analytically find the derivative.

### 3.5.9 **Exercise 8(b)**

For those impatient to use more advanced features of Julia, implement a version of Exercise 8(a) where $f_{\text{prime}}$ is calculated with auto-differentiation.

**In [33]:** `using ForwardDiff`

```julia
D(f) = x -> ForwardDiff.derivative(f, x)
```

**# example usage: create a function and get the derivative**
```julia
f(x) = x^2
f_prime = D(f)

f(0.1), f_prime(0.1)
```

**Out[33]:** `(0.01000000000000002, 0.2)`
1. Using the $D(f)$ operator definition above, implement a version of Newton’s method that does not require the user to provide an analytical derivative.

2. Test the sorts of $f$ functions which can be automatically integrated by ForwardDff.jl.

### 3.6 Solutions

#### 3.6.1 Exercise 1

In [34]:
```python
    function factorial2(n)
        k = 1
        for i in 1:n
            k *= i  # or k = k * i
        end
        return k
    end
```

factorial2(4)

Out[34]: 24

In [35]: factorial2(4) == factorial(4) # built-in function

Out[35]: true

#### 3.6.2 Exercise 2

In [36]:
```python
    function binomial_rv(n, p)
        count = 0
        U = rand(n)
        for i in 1:n
            if U[i] < p
                count += 1  # or count = count + 1
            end
        end
        return count
    end
```

for j in 1:25
    b = binomial_rv(10, 0.5)
    println("$b, ")
end

5, 6, 4, 4, 7, 5, 6, 5, 6, 8, 7, 4, 4, 6, 5, 6, 5, 4, 3, 5, 5, 6, 6, 4, 3,

#### 3.6.3 Exercise 3

Consider a circle with diameter 1 embedded in a 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 the point here 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.

In [37]: 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

3.143828

3.6.4 Exercise 4

In [38]: 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("\npayoff = $payoff")

Count = 1201230100
payoff = 1

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

In [39]: a = 1 < 2 ? "foo" : "bar"

Out[39]: "foo"


In [40]: a = 1 > 2 ? "foo" : "bar"

Out[40]: "bar"

Using this construction:

In [41]: payoff = 0.0
count = 0.0

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
println("\npayoff = $payoff")

Count = 1.0001230012
payoff = 1

3.6.5 Exercise 5

Here’s one solution

In [42]: using Plots
    gr(fmt=:png); # setting for easier display in jupyter notebooks
    α = 0.9
    n = 200
    x = zeros(n + 1)

    for t in 1:n
        x[t+1] = α * x[t] + randn()
    end
    plot(x)

Out[42]:
3.6.6 Exercise 6

In [43]: αs = [0.0, 0.8, 0.98]
   n = 200
   p = plot() # naming a plot to add to

   for α in αs
       x = zeros(n + 1)
       x[1] = 0.0
       for t in 1:n
           x[t+1] = α * x[t] + randn()
       end
       plot!(p, x, label = "alpha = \$α\") # add to plot p
   end
   p # display plot

Out[43]:
3.6.7 Exercise 7: Hint

As a hint, notice the following pattern for finding the number of draws of a uniform random number until it is below a given threshold

\[\text{In [44]}: \text{function drawsuntilthreshold(threshold; maxdraws=100)}\]
\[\text{for } i \text{ in } 1:\text{maxdraws} \]
\[\text{val} = \text{rand}() \]
\[\text{if } \text{val} < \text{threshold} \# \text{checks threshold} \]
\[\text{return } i \# \text{leaves function, returning draw number} \]
\[\text{end} \]
\[\text{return } \text{Inf} \# \text{if here, reached maxdraws} \]
\[\text{end} \]
\[\text{draws} = \text{drawsuntilthreshold(0.2, maxdraws=100)} \]

\[\text{Out[44]}: 2 \]

Additionally, it is sometimes convenient to add to just push numbers onto an array without indexing it directly

\[\text{In [45]}: \text{vals} = \text{zeros(0)} \# \text{empty vector} \]
\[\text{for } i \text{ in } 1:100 \]
\[\text{val} = \text{rand}() \]
\[\text{if } \text{val} < 0.5 \]
\[\text{push!}(\text{vals}, \text{val}) \]
\[\text{end} \]
\[\text{println("There were $(\text{length(vals)}) below 0.5")}\]
There were 51 below 0.5
Chapter 4

Julia Essentials

4.1 Contents

- Overview 4.2
- Common Data Types 4.3
- Iterating 4.4
- Comparisons and Logical Operators 4.5
- User-Defined Functions 4.6
- Broadcasting 4.7
- Scoping and Closures 4.8
- Exercises 4.9
- Solutions 4.10

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

4.2 Overview

Topics:

- Common data types
- Iteration
- More on user-defined functions
- Comparisons and logic

4.2.1 Setup

In [1]: using InstantiateFromURL
   # optionally add arguments to force installation: instantiate = true,
   # precompile = true
   github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

In [2]: using LinearAlgebra, Statistics
4.3 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.

4.3.1 Primitive Data Types

A particularly simple data type is a Boolean value, which can be either \texttt{true} or \texttt{false}.

\begin{verbatim}
In [3]: x = true
Out[3]: true

In [4]: typeof(x)
Out[4]: Bool

In [5]: y = 1 > 2  # now y = false
Out[5]: false
\end{verbatim}

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)

\begin{verbatim}
In [6]: typeof(1.0)
Out[6]: Float64

In [7]: typeof(1)
Out[7]: Int64
\end{verbatim}

If you’re running a 32 bit system you’ll still see \texttt{Float64}, but you will see \texttt{Int32} instead of \texttt{Int64} (see the section on Integer types from the Julia manual).

Arithmetic operations are fairly standard.

\begin{verbatim}
In [8]: x = 2; y = 1.0;
\end{verbatim}

The ; can be used to suppress output from a line of code, or to combine two lines of code together (as above), but is otherwise not necessary.
4.3. COMMON DATA TYPES

In [9]: x * y

Out[9]: 2.0

In [10]: x^2

Out[10]: 4

In [11]: y / x

Out[11]: 0.5

Although the * can be omitted for multiplication between a numeric literal and a variable.

In [12]: 2x - 3y

Out[12]: 1.0

A useful tool for displaying both expressions and code is to use the @show macro, which displays the text and the results.

In [13]: @show 2x - 3y
    @show x + y;

    2x - 3y = 1.0
    x + y = 3.0

Here we have used ; to suppress the output on the last line, which otherwise returns the results of x + y.

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

In [14]: x = 1 + 2im

Out[14]: 1 + 2im

In [15]: y = 1 - 2im

Out[15]: 1 - 2im

In [16]: x * y  # complex multiplication

Out[16]: 5 + 0im

There are several more primitive data types that we’ll introduce as necessary.
4.3.2 Strings

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

In Julia, strings are created using double quotation marks (single quotations are reserved for the character type).

In [17]: \texttt{x = "foobar"}

Out[17]: "foobar"

In [18]: \texttt{typeof(x)}

Out[18]: \texttt{String}

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

In [19]: \texttt{x = 10; y = 20}

Out[19]: 20

The \texttt{$} inside of a string is used to interpolate a variable.

In [20]: \texttt{"x = $x"}

Out[20]: "x = 10"

With parentheses, you can splice the results of expressions into strings as well.

In [21]: \texttt{"x + y = $(x + y)"}

Out[21]: "x + y = 30"

To concatenate strings use *.

In [22]: \texttt{"foo" * "bar"}

Out[22]: "foobar"

Julia provides many functions for working with strings.

In [23]: \texttt{s = "Charlie don't surf"}

Out[23]: "Charlie don't surf"

In [24]: \texttt{split(s)}
4.3. **COMMON DATA TYPES**

Out[24]: 3-element Array{SubString{String},1}:
   "Charlie"
   "don't"
   "surf"

In [25]: replace(s, "surf" => "ski")

Out[25]: "Charlie don't ski"

In [26]: split("fee,fi,fo", ",")

Out[26]: 3-element Array{SubString{String},1}:
   "fee"
   "fi"
   "fo"

In [27]: strip(" foobar ")  # remove whitespace

Out[27]: "foobar"

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

In [28]: match(r"(\d+)", "Top 10")  # find digits in string

Out[28]: RegexMatch("10", 1="10")

### 4.3.3 Containers

Julia has several basic types for storing collections of data.

We have already discussed arrays.

A related data type is a **tuple**, which is immutable and can contain different types.

In [29]: x = ("foo", "bar")
   y = ("foo", 2)

Out[29]: ("foo", 2)

In [30]: typeof(x), typeof(y)

Out[30]: (Tuple{String,String}, Tuple{String,Int64})

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

In particular, tuples do not support item assignment (i.e. `x[1] = "test"` would fail).

Tuples can be constructed with or without parentheses.

In [31]: x = "foo", 1
```
In [32]: function f()
    return "foo", 1
end
f()
Out[32]: ("foo", 1)

Tuples can also be unpacked directly into variables.

In [33]: x = ("foo", 1)
Out[33]: ("foo", 1)

In [34]: word, val = x
   println("word = $word, val = $val")

word = foo, val = 1

Tuples can be created with a hanging , – this is useful to create a tuple with one element.

In [35]: x = ("foo", 1,)
y = ("foo",)
typeof(x), typeof(y)
Out[35]: (Tuple{String,Int64}, Tuple{String})

Referencing Items

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

In [36]: x = [10, 20, 30, 40]
Out[36]: 4-element Array{Int64,1}:
  10  
  20  
  30  
  40  

In [37]: x[end]
Out[37]: 40

In [38]: x[end-1]
Out[38]: 30
```
4.3. COMMON DATA TYPES

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

```python
In [39]: x[1:3]
```

```
Out[39]: 3-element Array{Int64,1}:
   10
   20
   30
```

```python
In [40]: x[2:end]
```

```
Out[40]: 3-element Array{Int64,1}:
   20
   30
   40
```

The same slice notation works on strings.

```python
In [41]: "foobar"[3:end]
```

```
Out[41]: "obar"
```

**Dictionaries**

Another container type worth mentioning is dictionaries.

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

```python
In [42]: d = Dict("name" => "Frodo", "age" => 33)
```

```
Out[42]: Dict{String,Any} with 2 entries:
    "name" => "Frodo"
    "age"  => 33
```

```python
In [43]: d["age"]
```

```
Out[43]: 33
```

The strings name and age are called the keys.

The keys are mapped to values (in this case "Frodo" and 33).

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

**Note** Unlike in Python and some other dynamic languages, dictionaries are rarely the right approach (ie. often referred to as “the devil’s datastructure”).

The flexibility (i.e. can store anything and use anything as a key) frequently comes at the cost of performance if misused.

It is usually better to have collections of parameters and results in a named tuple, which both provide the compiler with more opportunities to optimize the performance, and also makes the code more safe.
4.4 Iterating

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

Julia provides neat and flexible tools for iteration as we now discuss.

4.4.1 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
In [44]: actions = ["surf", "ski"]
for action in actions
    println("Charlie doesn't $action")
end

Charlie doesn't surf
Charlie doesn't ski
```

They also include so-called iterators. You've already come across these types of values.

```julia
In [45]: for i in 1:3
    print(i)
end

1 2 3
```

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

```julia
In [46]: d = Dict("name" => "Frodo", "age" => 33)
Out[46]: Dict{String,Any} with 2 entries:
    "name" => "Frodo"
    "age"  => 33

In [47]: keys(d)
Out[47]: Base.KeySet 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()`.
4.4. ITERATING

In [48]: collect(keys(d))

Out[48]: 2-element Array{String,1}:
    "name"
    "age"

4.4.2 Looping without Indices

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

For example compare

In [49]: x_values = 1:5

Out[49]: 1:5

In [50]: for x in x_values
    println(x * x)
end

1
4
9
16
25

In [51]: for i in eachindex(x_values)
    println(x_values[i] * x_values[i])
end

1
4
9
16
25

Julia provides some functional-style helper functions (similar to Python and R) 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

In [52]: 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
In [53]: 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
```

### 4.4.3 Comprehensions

(See comprehensions documentation)

Comprehensions are an elegant tool for creating new arrays, dictionaries, etc. from iterables.

Here are some examples

```julia
In [54]: doubles = [ 2i for i in 1:4 ]
Out[54]: 4-element Array{Int64,1}:
          2
          4
          6
          8

In [55]: animals = ["dog", "cat", "bird"];  # Semicolon suppresses output
In [56]: plurals = [ animal "s" for animal in animals ]
Out[56]: 3-element Array{String,1}:
          "dogs"
          "cats"
          "birds"

In [57]: [ i + j for i in 1:3, j in 4:6 ]
Out[57]: 3×3 Array{Int64,2}:
          5  6  7
          6  7  8
          7  8  9

In [58]: [ i + j + k for i in 1:3, j in 4:6, k in 7:9 ]
Out[58]: 3×3×3 Array{Int64,3}:
          [:, :, 1] =
             12  13  14
Comprehensions can also create arrays of tuples or named tuples

In [59]: [ (i, j) for i in 1:2, j in animals]

Out[59]: 2×3 Array{Tuple{Int64,String},2}:
    (1, "dog") (1, "cat") (1, "bird")
    (2, "dog") (2, "cat") (2, "bird")

In [60]: [ (num = i, animal = j) for i in 1:2, j in animals]

Out[60]: 2×3 Array{NamedTuple{(:num, :animal),Tuple{Int64,String}},2}:
    (num = 1, animal = "dog") ... (num = 1, animal = "bird")
    (num = 2, animal = "dog") (num = 2, animal = "bird")

### 4.4.4 Generators

(See generator documentation)

In some cases, you may wish to use a comprehension to create an iterable list rather than actually making it a concrete array.

The benefit of this is that you can use functions which take general iterators rather than arrays without allocating and storing any temporary values.

For example, the following code generates a temporary array of size 10,000 and finds the sum.

In [61]: xs = 1:10000
    f(x) = x^2
    f_x = f.(xs)
    sum(f_x)

Out[61]: 333383335000

We could have created the temporary using a comprehension, or even done the comprehension within the sum function, but these all create temporary arrays.
CHAPTER 4. JULIA ESSENTIALS

\[
\begin{align*}
\text{sum}(f_x) &= 333383335000 \\
\text{sum}([f(x) \text{ for } x = xs]) &= 333383335000
\end{align*}
\]

Note, that if you were hand-code this, you would be able to calculate the sum by simply iterating to 10000, applying \( f \) to each number, and accumulating the results. No temporary vectors would be necessary.

A generator can emulate this behavior, leading to clear (and sometimes more efficient) code when used with any function that accepts iterators. All you need to do is drop the \( ] \) brackets.

In [63]: \( \text{sum}(f(x) \text{ for } x \text{ in } xs) \)

Out[63]: 333383335000

We can use \texttt{BenchmarkTools} to investigate

In [64]: \texttt{using BenchmarkTools}
\texttt{@btime sum([f(x) \text{ for } x \text{ in } $xs$])}
\texttt{@btime sum(f.($xs$))}
\texttt{@btime sum(f(x) \text{ for } x \text{ in } $xs$)};

\[
\begin{align*}
10.138 \mu s & \ (2 \text{ allocations: 78.20 KiB}) \\
9.508 \mu s & \ (2 \text{ allocations: 78.20 KiB}) \\
4.725 \text{ ns} & \ (0 \text{ allocations: 0 bytes})
\end{align*}
\]

Notice that the first two cases are nearly identical, and allocate a temporary array, while the final case using generators has no allocations.

In this example you may see a speedup of over 1000x. Whether using generators leads to code that is faster or slower depends on the circumstances, and you should (1) always profile rather than guess; and (2) worry about code clarity first, and performance second—if ever.

4.5 Comparisons and Logical Operators

4.5.1 Comparisons

As we saw earlier, when testing for equality we use \( == \).

In [65]: \( x = 1 \)

Out[65]: 1

In [66]: \( x == 2 \)

Out[66]: \texttt{false}

For “not equal” use \( != \) or \( \texttt{\textbackslash ne<TAB>} \).
4.6. USER-DEFINED FUNCTIONS

In [67]: x != 3

Out[67]: true

Julia can also test approximate equality with ≈ (\approx<TAB>).

In [68]: 1 + 1E-8 ≈ 1

Out[68]: true

Be careful when using this, however, as there are subtleties involving the scales of the quantities compared.

4.5.2 Combining Expressions

Here are the standard logical connectives (conjunction, disjunction)

In [69]: true && false

Out[69]: false

In [70]: true || false

Out[70]: true

Remember

• P && Q is true if both are true, otherwise it’s false.
• P || Q is false if both are false, otherwise it’s true.

4.6 User-Defined Functions

Let’s talk a little more about user-defined functions.

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

• separating different strands of logic
• facilitating code reuse (writing the same thing twice is always a bad idea)

Julia functions are convenient:

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

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

For now let’s just cover some of the different ways of defining functions.
4.6.1 Return Statement

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

In [71]:
```julia
function f1(a, b)
    return a * b
end

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

Out[71]: f2 (generic function with 1 method)

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:

In [72]:
```julia
function foo(x)
    if x > 0
        return "positive"
    end
    return "nonpositive"
end
```

Out[72]: foo (generic function with 1 method)

4.6.2 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`:

In [73]:
```julia
f(x) = sin(1 / x)
```

Out[73]: f (generic function with 2 methods)

Let’s check that it works:

In [74]:
```julia
f(1 / pi)
```

Out[74]: 1.2246467991473532e-16
4.6. USER-DEFINED FUNCTIONS

Julia also allows you to define anonymous functions.
For example, to define \( f(x) = \sin(1 / x) \) you can use \( x \to \sin(1 / x) \).
The difference is that the second function has no name bound to it.
How can you use a function with no name?
Typically it’s as an argument to another function

In [75]: map(x \to \sin(1 / x), randn(3))  # apply function to each element

Out[75]: 3-element Array{Float64,1}:
-0.994378709150385
0.7959953516509257
0.7931544866509989

4.6.3 Optional and Keyword Arguments

(See keyword arguments documentation)
Function arguments can be given default values

In [76]: f(x, a = 1) = exp(cos(a \times x))

Out[76]: f (generic function with 3 methods)

If the argument is not supplied, the default value is substituted.

In [77]: f(pi)

Out[77]: 0.36787944117144233

In [78]: f(pi, 2)

Out[78]: 2.718281828459045

Another option is to use keyword arguments.
The difference between keyword and standard (positional) arguments is that they are parsed and bounded by name rather than the order in the function call.
For example, in the call

In [79]: f(x; a = 1) = exp(cos(a \times x))  # note the ; in the definition
   f(pi, a = 2)  # calling with ; is usually optional and generally discouraged

Out[79]: 2.718281828459045
4.7 Broadcasting

(See broadcasting documentation)

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 an iterable \( x \_vec \) to produce a new result \( y \_vec \)

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 \( x \_vec = [2.0, 4.0, 6.0, 8.0] \).

The following code will do the job

\[
\text{In [80]:} \quad \text{x}\_vec = [2.0, 4.0, 6.0, 8.0] \\
\text{y}\_vec = \text{similar(x}\_vec) \\
\text{for } (i, x) \text{ in enumerate(x}\_vec) \\
\quad \text{y}\_vec[i] = \sin(x) \\
\text{end}
\]

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

\[
\text{In [81]:} \quad \text{y}\_vec = \sin.(x}\_vec)
\]

\[
\text{Out[81]:} \quad 4\text{-element Array}\{\text{Float64,1}\}: \\
\quad 0.9092974268256817 \\
\quad -0.7568024953079282 \\
\quad -0.27941549819892586 \\
\quad 0.9893582466233818
\]

More generally, if \( f \) is any Julia function, then \( f. \) references the broadcasted 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’ll 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.

\[
\text{In [82]:} \quad \text{function } \text{chisq}(k) \\
\quad \text{@assert } k > 0 \\
\quad z = \text{randn}(k) \\
\quad \text{return } \text{sum}(z -\rightarrow z^2, z) \quad \text{# same as `sum(x^2 for x in z)`}
\]

\[
\text{Out[82]:} \quad \text{chisq \ (generic function with 1 method)}
\]

The macro \text{@assert} will check that the next expression evaluates to \text{true}, and will stop and display an error otherwise.

\[
\text{In [83]:} \quad \text{chisq(3)}
\]
Note that calls with integers less than 1 will trigger an assertion failure inside the function body.

In [84]: chisq(-2)

AssertionError: k > 0

Stacktrace:
[1] chisq(::Int64) at ./In[82]:2
[2] top-level scope at In[84]:1

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

In [85]: chisq([2, 4, 6])

Out[85]: 3-element Array{Float64,1}:
       6.478871317138532
       5.419973596174483
       6.02040608718741

The broadcasting notation is not simply vectorization, as it is able to “fuse” multiple broadcasts together to generate efficient code.

In [86]: x = 1.0:1.0:5.0
    y = [2.0, 4.0, 5.0, 6.0, 8.0]
    z = similar(y)
    z .= x .+ y .- sin.(x)  # generates efficient code instead of many temporaries

Out[86]: 5-element Array{Float64,1}:
          2.1585290151921033
          5.090702573174318
          7.858879991940133
         10.756802495307928
         13.958924274663138

A convenience macro for adding broadcasting on every function call is @.

In [87]: @. z = x + y - sin(x)

Out[87]: 5-element Array{Float64,1}:
          2.1585290151921033
          5.090702573174318
          7.858879991940133
         10.756802495307928
         13.958924274663138
Since the +, -, = operators are functions, behind the scenes this is broadcasting against both the x and y vectors.

The compiler will fix anything which is a scalar, and otherwise iterate across every vector

In [88]:
```julia
f(a, b) = a + b # bivariate function
a = [1 2 3]
b = [4 5 6]
@show f.(a, b) # across both
@show f.(a, 2); # fix scalar for second
```

```
f.(a, b) = [5 7 9]
f.(a, 2) = [3 4 5]
```

The compiler is only able to detect “scalar” values in this way for a limited number of types (e.g. integers, floating points, etc) and some packages (e.g. Distributions).

For other types, you will need to wrap any scalars in \texttt{Ref} to fix them, or else it will try to broadcast the value.

Another place that you may use a \texttt{Ref} is to fix a function parameter you do not want to broadcast over.

In [89]:
```julia
f(x, y) = [1, 2, 3] \cdot x + y # "\cdot" can be typed by \texttt{\textbackslash cdot}<tab>
f([3, 4, 5], 2) # uses vector as first parameter
f(\texttt{Ref}([3, 4, 5]), [2, 3]) # broadcasting over 2nd parameter, fixing first
```

Out[89]:
```
2-element Array{Int64,1}:
    28
    29
```

### 4.8 Scoping and Closures

Since global variables are usually a bad idea, we will concentrate on understanding the role of good local scoping practice.

That said, while many of the variables in these Jupyter notebook are global, we have been careful to write the code so that the entire code could be copied inside of a function.

When copied inside a function, variables become local and functions become closures.

\textbf{Warning}.

For/while loops and global variables in Jupyter vs. the REPL: * In the current version of Julia, there is a distinction between the use of scope in an interactive Jupyter environment. * The description here of globals applies to Jupyter notebooks, and may also apply to the REPL and top-level scripts. * In general, you should be creating functions when working with .jl files, and the distinction generally won’t apply.

For more information on using globals outside of Jupyter, (see variable scoping documentation), though these rules are likely to become consistent in a future version.
4.8. SCOPING AND CLOSURES

4.8.1 Functions

The scope of a variable name determines where it is valid to refer to it, and how clashes between names can occur.

Think of the scope as a list of all of the name bindings of relevant variables.

Different scopes could contain the same name but be assigned to different things.

An obvious place to start is to notice that functions introduce their own local names.

In [90]: f(x) = x^2  # local `x` in scope
   # x is not bound to anything in this outer scope
   y = 5
   f(y)

Out[90]: 25

This would be roughly equivalent to

In [91]: function g()  # scope within the `g` function
   f(x) = x^2  # local `x` in scope
   # x is not bound to anything in this outer scope
   y = 5
   f(y)
   end
   g()  # run the function

Out[91]: 25

This is also equivalent if the y was changed to x, since it is a different scope.

In [92]: f(x) = x^2  # local `x` in scope
   # x is not bound to anything in this outer scope
   x = 5  # a different `x` than the local variable name
   f(x)  # calling `f` with `x`

Out[92]: 25

The scoping also applies to named arguments in functions.

In [93]: f(x; y = 1) = x + y  # `x` and `y` are names local to the `f` function
   xval = 0.1
   yval = 2
   f(xval; y = yval)

Out[93]: 2.1

Due to scoping, you could write this as
In [94]: f(x; y = 1) = x + y  # `x` and `y` are names local to the `f` function
x = 0.1
y = 2
f(x; y = y)  # left hand `y` is the local name of the argument in the function

Out[94]: 2.1

Similarly to named arguments, the local scope also works with named tuples.

In [95]: xval = 0.1
   yval = 2
  @show (x = xval, y = yval)  # named tuple with names `x` and `y`
   x = 0.1
   y = 2
   # create a named tuple with names `x` and `y` local to the tuple, bound to the RHS `x`
   and `y`
   (x = x, y = y)
   (x = xval, y = yval) = (x = 0.1, y = 2)

Out[95]: (x = 0.1, y = 2)

As you use Julia, you will find that scoping is very natural and that there is no reason to avoid using `x` and `y` in both places.

In fact, it frequently leads to clear code closer to the math when you don’t need to specify intermediaries.

Another example is with broadcasting

In [96]: f(x) = x^2  # local `x` in scope
   x = 1:5  # not an integer
   f.(x)  # broadcasts the x^2 function over the vector

Out[96]: 5-element Array{Int64,1}:
   1
   4
   9
  16
  25

4.8.2 Closures

Frequently, you will want to have a function that calculates a value given some fixed parameters.
4.8. SCOPING AND CLOSURES

In [97]:
   \[ f(x, a) = a \times x^2 \]
   
   \[ f(1, 0.2) \]

Out[97]: 0.2

While the above was convenient, there are other times when you want to simply fix a variable or refer to something already calculated.

In [98]:
   \[ a = 0.2 \]
   \[ f(x) = a \times x^2 \quad \# \text{refers to the `a` in the outer scope} \]
   \[ f(1) \quad \# \text{univariate function} \]

Out[98]: 0.2

When the function \( f \) is parsed in Julia, it will look to see if any of the variables are already defined in the current scope.

In this case, it finds the \( a \) since it was defined previously, whereas if the code defines \( a = 0.2 \) after the \( f(x) \) definition, it would fail.

This also works when embedded in other functions

In [99]:
   \[
   \text{function } g(a)
   f(x) = a \times x^2 \quad \# \text{refers to the `a` passed in the function}
   f(1) \quad \# \text{univariate function}
   \text{end}
   \]
   
   \[ g(0.2) \]

Out[99]: 0.2

Comparing the two: the key here is not that \( a \) is a global variable, but rather that the \( f \) function is defined to capture a variable from an outer scope.

This is called a closure, and are used throughout the lectures.

It is generally bad practice to modify the captured variable in the function, but otherwise the code becomes very clear.

One place where this can be helpful is in a string of dependent calculations.

For example, if you wanted to calculate \( (a, b, c) \) from \( a = f(x), b = g(a), c = h(a, b) \)
where \( f(x) = x^2, g(a) = 2a, h(a, b) = a + b \)

In [100]:
   \[
   \text{function } solvemodel(x)
   a = x^2
   b = 2 \times a
   c = a + b
   \text{return } (a = a, b = b, c = c) \quad \# \text{note local scope of tuples!}
   \text{end}
   \]
   
   \[ solvemodel(0.1) \]

Out[100]:\[(a = 0.01000000000000002, b = 0.02000000000000004, c = 0.03000000000000006)\]
### 4.8.3 Higher-Order Functions

One of the benefits of working with closures and functions is that you can return them from other functions.

This leads to some natural programming patterns we have already been using, where we can use **functions of functions** and **functions returning functions** (or closures).

To see a simple example, consider functions that accept other functions (including closures)

```julia
In [101]: twice(f, x) = f(f(x))  # applies f to itself twice
    f(x) = x^2
   @show twice(f, 2.0)
    twice(x -> x^2, 2.0)  
    a = 5
    g(x) = a * x
    @show twice(g, 2.0);  # using a closure

    twice(f, 2.0) = 16.0
    twice(g, 2.0) = 50.0
```

This pattern has already been used extensively in our code and is key to keeping things like interpolation, numerical integration, and plotting generic.

One example of using this in a library is Expectations.jl, where we can pass a function to the `expectation` function.

```julia
In [102]: using Expectations, Distributions

    @show d = Exponential(2.0)
    f(x) = x^2
    @show expectation(f, d);  # E(f(x))

    d = Exponential(2.0) = Exponential{Float64}(@=2.0)
    expectation(f, d) = 8.00000000000004
```

Another example is for a function that returns a closure itself.

```julia
In [103]: function multiplyit(a, g)
    return x -> a * g(x)  # function with `g` used in the closure
end

    f(x) = x^2
    h = multiplyit(2.0, f)  # use our quadratic, returns a new function which doubles the
    result
    h(2)  # returned function is like any other function

Out[103]: 8.0
```

You can create and define using `function` as well
In [104]:

```python
function snapabove(g, a)
    function f(x)
        if x > a
            return g(x)  # "a" is captured in the closure f
        else
            return g(a)
        end
    end
    return f  # closure with the embedded a
end

f(x) = x^2
h = snapabove(f, 2.0)

using Plots
gr(fmt=:png);
plot(h, 0.0:0.1:3.0)
```

Out[104]:

```
```

### 4.8.4 Loops

The `for` and `while` loops also introduce a local scope, and you can roughly reason about them the same way you would a function/closure.

In particular

In [105]:

```python
for i in 1:2  # introduces local i
dval1 = i
    println(i)
end
```
# @show (i, dval1) # would fail as neither exists in this scope

```julia
for i in 1:2  # introduces a different local i
    println(i)
end
```

```
1
2
1
2
```

On the other hand just as with closures, if a variable is already defined it will be available in
the inner scope.

```
In [106]: dval2 = 0  # introduces variables
    for i in 1:2  # introduces local i
        dval2 = i  # refers to outer variable
    end
    dval2  # still can't refer to `i`
```

```
Out[106]: 2
```

Similarly, for while loops

```
In [107]: val = 1.0
tol = 0.002
while val > tol
    old = val
    val = val / 2
    difference = val - old
end

@show val;
# @show difference fails, not in scope
```

```
val = 0.001953125
```

### 4.8.5 A Quick Check for Scoping Design

While we have argued against global variables as poor practice, you may have noticed that in
Jupyter notebooks we have been using them throughout.

Here, global variables are used in an interactive editor because they are convenient, and not
because they are essential to the design of functions.

A simple test of the difference is to take a segment of code and wrap it in a function, for ex-
ample

```
In [108]: x = 2.0
    f(y) = x + y
```
Here, the $x$ and $z$ are global variables, the function $f$ refers to the global variable $x$, and the global variable $z$ is modified in the `for` loop.

However, you can simply wrap the entire code in a function

```python
In [109]:

def wrapped():
    x = 2.0
    f(y) = x + y
    z = f(4.0)

    for i in 1:3
        z += i
    end

    println("z = $z")
end

wrapped()
```

Now, there are no global variables.

While it is convenient to skip wrapping our code throughout, in general you will want to wrap any performance sensitive code in this way.

### 4.9 Exercises

#### 4.9.1 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: `iseven` returns `true` for even numbers and `false` for odds.

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.

#### 4.9.2 Exercise 2

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

(1)

Using \texttt{enumerate()} in your loop, write a function \texttt{p} such that \texttt{p(x, coeff)} computes the value in (1) given a point \texttt{x} and an array of coefficients \texttt{coeff}.

4.9.3 Exercise 3

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

Hint: \texttt{toUpperCase("foo")} returns "FOO".

4.9.4 Exercise 4

Write a function that takes two sequences \texttt{seq\_a} and \texttt{seq\_b} as arguments and returns \texttt{true} if every element in \texttt{seq\_a} is also an element of \texttt{seq\_b}, else \texttt{false}.

- By “sequence” we mean an array, tuple or string.

4.9.5 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 \texttt{linapprox} that takes as arguments

- A function \texttt{f} mapping some interval \([a, b]\) into \(\mathbb{R}\).
- Two scalars \texttt{a} and \texttt{b} providing the limits of this interval.
- An integer \texttt{n} determining the number of grid points.
- A number \texttt{x} satisfying \(a \leq x \leq b\).

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

Aim for clarity, not efficiency.

Hint: use the function \texttt{range} to linearly space numbers.

4.9.6 Exercise 6

The following data lists US cities and their populations.

Copy this text into a text file called \texttt{us\_cities.txt} and save it in your present working directory.

- That is, save it in the location Julia returns when you call \texttt{pwd()}.

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

```julia
In [110]: open("us_cities.txt", "w") do f
    write(f,
"new york: 8244910
```
4.10. SOLUTIONS

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

end

Out[110]: 167

Write a program to calculate total population across these cities.

Hints:

• If f is a file type 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.

4.9.7 Exercise 7

Redo Exercise 5 except

1. Pass in a range instead of the a, b, and n. Test with a range such as nodes = -1.0:0.5:1.0.
2. Instead of the while used in the solution to Exercise 5, find a better way to efficiently bracket the x in the nodes.

Hints: * Rather than the signature as function linapprox(f, a, b, n, x), it should be called as function linapprox(f, nodes, x). * step(nodes), length(nodes), nodes[1], and nodes[end] may be useful. * Type ?÷ into jupyter to explore quotients from Euclidean division for more efficient bracketing.

4.10 Solutions

4.10.1 Exercise 1

Part 1 solution:

Here’s one possible solution

In [111]: x_vals = [1, 2, 3]
y_vals = [1, 1, 1]
sum(x * y for (x, y) in zip(x_vals, y_vals))

Out[111]: 6

Part 2 solution:

One solution is
In [112]: `sum(iseven, 0:99)`

Out[112]: 50

Part 3 solution:

Here’s one possibility

In [113]: `pairs = ((2, 5), (4, 2), (9, 8), (12, 10))
   sum(xy -> all(iseven, xy), pairs)`

Out[113]: 2

4.10.2 Exercise 2

In [114]: `p(x, coeff) = sum(a * x^(i-1) for (i, a) in enumerate(coeff))`

Out[114]: `p (generic function with 1 method)`

In [115]: `p(1, (2, 4))`

Out[115]: 6

4.10.3 Exercise 3

Here’s one solutions:

In [116]: `function f_ex3(string)
   count = 0
   for letter in string
       if (letter == uppercase(letter)) && isletter(letter)
           count += 1
   end
   return count
end`

f_ex3("The Rain in Spain")

Out[116]: 3

4.10.4 Exercise 4

Here’s one solutions:

In [117]: `function f_ex4(seq_a, seq_b)
   is_subset = true
   for a in seq_a
       if a ∉ seq_b
           is_subset = false
   end
   return is_subset
end`

Out[117]: `false`
4.10. SOLUTIONS

```python
is_subset = ...

# 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

In [118]: f_ex4_2(seq_a, seq_b) = Set(seq_a) \ Set(seq_b) # \subseteq (\textsubscript{\textup{i}}) is
println(f_ex4_2([[1, 2], [1, 2, 3]]))
println(f_ex4_2([[1, 2, 3], [1, 2]]))

true
false

4.10.5 Exercise 5

In [119]: 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
```

Out[119]: linapprox (generic function with 1 method)

Let's test it

In [120]: f_ex5(x) = x^2
g_ex5(x) = linapprox(f_ex5, -1, 1, 3, x)
4.10.6 Exercise 6

In [122]: 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
Chapter 5

Arrays, Tuples, Ranges, and Other Fundamental Types

5.1 Contents

- Overview 5.2
- Array Basics 5.3
- Operations on Arrays 5.4
- Ranges 5.5
- Tuples and Named Tuples 5.6
- Nothing, Missing, and Unions 5.7
- Exercises 5.8
- Solutions 5.9

“Let’s be clear: the work of science has nothing whatever to do with consensus. Consensus is the business of politics. Science, on the contrary, requires only one investigator who happens to be right, which means that he or she has results that are verifiable by reference to the real world. In science consensus is irrelevant. What is relevant is reproducible results.” – Michael Crichton

5.2 Overview

In Julia, arrays and tuples are the most important data type for working with numerical data.

In this lecture we give more details on

- creating and manipulating Julia arrays
- fundamental array processing operations
- basic matrix algebra
- tuples and named tuples
- ranges
- nothing, missing, and unions
5.2.1 Setup

In [1]: using InstantiateFromURL
   # optionally add arguments to force installation: instantiate = true,
   # precompile = true
   github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

In [2]: using LinearAlgebra, Statistics

5.3 Array Basics

(See multi-dimensional arrays documentation)

Since it is one of the most important types, we will start with arrays.
Later, we will see how arrays (and all other types in Julia) are handled in a generic and extensible way.

5.3.1 Shape and Dimension

We’ve already seen some Julia arrays in action

In [3]: a = [10, 20, 30]

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

In [4]: a = [1.0, 2.0, 3.0]

Out[4]: 3-element Array{Float64,1}:
   1.0
   2.0
   3.0

The output tells us that the arrays are of types Array{Int64,1} and Array{Float64,1} respectively.
Here Int64 and Float64 are types for the elements inferred by the compiler.
We’ll talk more about types later.
The 1 in Array{Int64,1} and Array{Any,1} indicates that the array is one dimensional (i.e., a Vector).
This is the default for many Julia functions that create arrays

In [5]: typeof(randn(100))

Out[5]: Array{Float64,1}
5.3. ARRAY BASICS

In Julia, one dimensional vectors are best interpreted as column vectors, which we will see when we take transposes.

We can check the dimensions of \(a\) using `size()` and `ndims()` functions

```
In [6]: ndims(a)
Out[6]: 1

In [7]: size(a)
Out[7]: (3,)
```

The syntax `(3,)` displays a tuple containing one element – the size along the one dimension that exists.

**Array vs Vector vs Matrix**

In Julia, `Vector` and `Matrix` are just aliases for one- and two-dimensional arrays respectively

```
In [8]: Array{Int64, 1} == Vector{Int64}
   Array{Int64, 2} == Matrix{Int64}

Out[8]: true
```

Vector construction with `,` is then interpreted as a column vector.

To see this, we can create a column vector and row vector more directly

```
In [9]: [1, 2, 3] == [1; 2; 3]  # both column vectors
Out[9]: true

In [10]: [1 2 3]  # a row vector is 2-dimensional
Out[10]: 1x3 Array{Int64, 2}:
          1  2  3
```

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 for matrix algebra.

- Multiplying by a row vector is different to multiplying by a column vector.

On the other, we use arrays in many settings that don’t involve matrix algebra.

In such cases, we don’t care about the distinction between row and column vectors.

This is why many Julia functions return flat arrays by default.
5.3.2 Creating Arrays

Functions that Create Arrays

We’ve already seen some functions for creating a vector filled with $0.0$

In [11]: `zeros(3)`

Out[11]: 3-element Array{Float64,1}:
   0.0
   0.0
   0.0

This generalizes to matrices and higher dimensional arrays

In [12]: `zeros(2, 2)`

Out[12]: 2×2 Array{Float64,2}:
   0.0  0.0
   0.0  0.0

To return an array filled with a single value, use `fill`

In [13]: `fill(5.0, 2, 2)`

Out[13]: 2×2 Array{Float64,2}:
   5.0  5.0
   5.0  5.0

Finally, you can create an empty array using the `Array()` constructor

In [14]: `x = Array{Float64}(undef, 2, 2)`

Out[14]: 2×2 Array{Float64,2}:
   6.91839e-310  6.91838e-310
   6.91839e-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)

If you need more control over the types, fill with a non-floating point

In [15]: `fill(0, 2, 2)`  # fills with 0, not 0.0

Out[15]: 2×2 Array{Int64,2}:
   0  0
   0  0

Or fill with a boolean type

In [16]: `fill(false, 2, 2)`  # produces a boolean matrix

Out[16]: 2×2 Array{Bool,2}:
   0  0
   0  0
Creating Arrays from Existing Arrays

For the most part, we will avoid directly specifying the types of arrays, and let the compiler deduce the optimal types on its own.

The reasons for this, discussed in more detail in this lecture, are to ensure both clarity and generality.

One place this can be inconvenient is when we need to create an array based on an existing array.

First, note that assignment in Julia binds a name to a value, but does not make a copy of that type

In [17]:
```
x = [1, 2, 3]
y = x
y[1] = 2
x
```

Out[17]: 3-element Array{Int64,1}:
2
2
3

In the above, \( y = x \) simply creates a new named binding called \( y \) which refers to whatever \( x \) currently binds to.

To copy the data, you need to be more explicit

In [18]:
```
x = [1, 2, 3]
y = copy(x)
y[1] = 2
x
```

Out[18]: 3-element Array{Int64,1}:
1
2
3

However, rather than making a copy of \( x \), you may want to just have a similarly sized array

In [19]:
```
x = [1, 2, 3]
y = similar(x)
y
```

Out[19]: 3-element Array{Int64,1}:
1
1
0

We can also use `similar` to pre-allocate a vector with a different size, but the same shape

In [20]:
```
x = [1, 2, 3]
y = similar(x, 4)  # make a vector of length 4
```
Which generalizes to higher dimensions

\[
\begin{align*}
\text{Out[20]}: & \quad \text{4-element Array}\{\text{Int64},1\} : \\
& \quad 140029121887504 \\
& \quad 140029758928704 \\
& \quad 140029448573616 \\
& \quad 140029694705664
\end{align*}
\]

Manual Array Definitions

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

\[
\begin{align*}
\text{In [21]}: & \quad x = [1, 2, 3] \\
& \quad y = \text{similar}(x, 2, 2) \quad \text{# make a 2x2 matrix}
\end{align*}
\]

\[
\begin{align*}
\text{Out[21]}: & \quad 2\times2 \text{ Array}\{\text{Int64},2\} : \\
& \quad 140029122207120 \quad 140029122207184 \\
& \quad 140029122207152 \quad 140029493746368
\end{align*}
\]

In two dimensions we can proceed as follows

\[
\begin{align*}
\text{In [22]}: & \quad a = [10, 20, 30, 40] \\
\text{Out[22]}: & \quad 4\text{-element Array}\{\text{Int64},1\} : \\
& \quad 10 \\
& \quad 20 \\
& \quad 30 \\
& \quad 40
\end{align*}
\]

\[
\begin{align*}
\text{In [23]}: & \quad a = [10 \ 20 \ 30 \ 40] \quad \text{# two dimensional, shape is 1 x n} \\
\text{Out[23]}: & \quad 1\times4 \text{ Array}\{\text{Int64},2\} : \\
& \quad 10 \ 20 \ 30 \ 40
\end{align*}
\]

\[
\begin{align*}
\text{In [24]}: & \quad \text{ndims}(a) \\
\text{Out[24]}: & \quad 2 \\
\text{In [25]}: & \quad a = [10 \ 20; \ 30 \ 40] \quad \text{# 2 x 2} \\
\text{Out[25]}: & \quad 2\times2 \text{ Array}\{\text{Int64},2\} : \\
& \quad 10 \ 20 \\
& \quad 30 \ 40
\end{align*}
\]

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

\[
\begin{align*}
\text{In [26]}: & \quad a = [10; 20; 30; 40]
\end{align*}
\]
5.3. **ARRAY BASICS**

**Out[26]:** 4-element Array{Int64,1}:
10
20
30
40

**In [27]:** ndims(a)

**Out[27]:** 1

Instead transpose the matrix (or adjoint if complex)

**In [28]:** a = [10 20 30 40]'

**Out[28]:** 4×1 Adjoint{Int64,Array{Int64,2}}:
10
20
30
40

**In [29]:** ndims(a)

**Out[29]:** 2

### 5.3.3 Array Indexing

We’ve already seen the basics of array indexing

**In [30]:** a = [10 20 30 40]
a[end-1]

**Out[30]:** 30

**In [31]:** a[1:3]

**Out[31]:** 3-element Array{Int64,1}:
10
20
30

For 2D arrays the index syntax is straightforward

**In [32]:** a = randn(2, 2)
a[1, 1]

**Out[32]:** 0.15765041580654648

**In [33]:** a[1, :]  # first row
Booleans can be used to extract elements

In [35]: a = randn(2, 2)

Out[35]:
2×2 Array{Float64,2}:
 0.616704  0.374922
 1.33094  -0.670024

In [36]: b = [true false; false true]

Out[36]:
2×2 Array{Bool,2}:
 1  0
 0  1

In [37]: a[b]

Out[37]:
2-element Array{Float64,1}:
 0.6167040565642389
 -0.6700242069255393

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

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

In [38]: a = zeros(4)

Out[38]:
4-element Array{Float64,1}:
 0.0
 0.0
 0.0
 0.0

In [39]: a[2:end] .= 42

Out[39]:
3-element view(::Array{Float64,1}, 2:4) with eltype Float64:
 42.0
 42.0
 42.0

In [40]: a

Out[40]:
4-element Array{Float64,1}:
 0.0
 42.0
 42.0
 42.0
5.3.4 Views and Slices

Using the : notation provides a slice of an array, copying the sub-array to a new array with a similar type.

In [41]:
\[
\begin{align*}
a &= \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \\
b &= a[:, 2] \\
\end{align*}
\]

@show b
\[
a[:, 2] = [4, 5] \quad \text{# modify a}
\]

@show a
@show b;

\[
b = [2, 4] \\
a = \begin{bmatrix} 1 & 4 \\ 3 & 5 \end{bmatrix} \\
b = [2, 4]
\]

A view on the other hand does not copy the value

In [42]:
\[
\begin{align*}
a &= \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \\
@views b &= a[:, 2] \\
@show b \\
a[:, 2] &= [4, 5] \\
@show a \\
@show b;
\end{align*}
\]

\[
b = [2, 4] \\
a = \begin{bmatrix} 1 & 4 \\ 3 & 5 \end{bmatrix} \\
b = [4, 5]
\]

Note that the only difference is the @views macro, which will replace any slices with views in the expression.

An alternative is to call the view function directly – though it is generally discouraged since it is a step away from the math.

In [43]:
\[
\begin{align*}
@views b &= a[:, 2] \\
\text{view(a, :, 2)} &= b
\end{align*}
\]

Out[43]: true

As with most programming in Julia, it is best to avoid prematurely assuming that @views will have a significant impact on performance, and stress code clarity above all else.

Another important lesson about @views is that they are not normal, dense arrays.

In [44]:
\[
\begin{align*}
a &= \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \\
b_slice &= a[:, 2] \\
@show typeof(b_slice) \\
@show typeof(a) \\
@views b &= a[:, 2] \\
@show typeof(b);
\end{align*}
\]
CHAPTER 5. ARRAYS, TUPLES, RANGES, AND OTHER FUNDAMENTAL TYPES

```plaintext
typeof(b_slice) = Array{Int64,1}
typeof(a) = Array{Int64,2}
typeof(b) = SubArray{Int64,1,Array{Int64,2},Tuple{Base.Slice{Base.OneTo{Int64}},Int64},true}
```

The type of \( \mathbf{b} \) is a good example of how types are not as they may seem.

Similarly

```plaintext
In [45]: a = [1 2; 3 4]
   ...: b = a'  # transpose
typeof(b)
```

```
Out[45]: Adjoint{Int64,Array{Int64,2}}
```

To copy into a dense array

```plaintext
In [46]: a = [1 2; 3 4]
   ...: b = a'  # transpose
   ...: c = Matrix(b)  # convert to matrix
   ...: d = collect(b)  # also `collect` works on any iterable
   ...: c == d
```

```
Out[46]: true
```

5.3.5 Special Matrices

As we saw with transpose, sometimes types that look like matrices are not stored as a dense array.

As an example, consider creating a diagonal matrix

```plaintext
In [47]: d = [1.0, 2.0]
   ...: a = Diagonal(d)
```

```
Out[47]: 2×2 Diagonal{Float64,Array{Float64,1}}:
   1.0   □
   □   2.0
```

As you can see, the type is 2×2 Diagonal{Float64,Array{Float64,1}}, which is not a 2-dimensional array.

The reasons for this are both efficiency in storage, as well as efficiency in arithmetic and matrix operations.

In every important sense, matrix types such as Diagonal are just as much a “matrix” as the dense matrices we have using (see the introduction to types lecture for more)

```plaintext
In [48]: @show 2a
   ...: b = rand(2,2)
   ...: @show b * a;
```

```
Out[48]:
```

```
5.3. ARRAY BASICS

\[ \begin{bmatrix} 2 \cdot 0 & 0 \\ 0 & 4 \cdot 0 \end{bmatrix} \]
\[
\begin{bmatrix} 0.07902796541012158 & 0.36525218581573515 \\ 0.5277291038116809 & 1.824094575599461 \end{bmatrix}
\]

Another example is in the construction of an identity matrix, where a naive implementation is

In [49]:
   b = [1.0 2.0; 3.0 4.0]
   b - Diagonal([1.0, 1.0])  # poor style, inefficient code

Out[49]: 2×2 Array{Float64,2}:
          0.0  2.0
          3.0  3.0

Whereas you should instead use

In [50]:
   b = [1.0 2.0; 3.0 4.0]
   b - I  # good style, and note the lack of dimensions of I

Out[50]: 2×2 Array{Float64,2}:
          0.0  2.0
          3.0  3.0

While the implementation of \( I \) is a little abstract to go into at this point, a hint is:

In [51]:
typeof(I)

Out[51]: UniformScaling{Bool}

This is a UniformScaling type rather than an identity matrix, making it much more powerful and general.

5.3.6 Assignment and Passing Arrays

As discussed above, in Julia, the left hand side of an assignment is a “binding” or a label to a value.

In [52]:
   x = [1 2 3]
   y = x  # name `y` binds to whatever value `x` bound to

Out[52]: 1×3 Array{Int64,2}:
          1  2  3

The consequence of this, is that you can re-bind that name.

In [53]:
   x = [1 2 3]
   y = x  # name `y` binds to whatever `x` bound to
   z = [2 3 4]
   y = z  # only changes name binding, not value!
  @show (x, y, z);
(x, y, z) = ([1 2 3], [2 3 4], [2 3 4])

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.

In the above, suppose you had meant to change the value of x to the values of y, you need to assign the values rather than the name.

In [54]: x = [1 2 3]
y = x  # name `y` binds to whatever `x` bound to
z = [2 3 4]
y .= z  # now dispatches the assignment of each element
@show (x, y, z);

(x, y, z) = ([2 3 4], [2 3 4], [2 3 4])

Alternatively, you could have used y[:] = z.

This applies to in-place functions as well.

First, define a simple function for a linear map

In [55]: function f(x)
    return [1 2; 3 4] * x  # matrix * column vector
end

val = [1, 2]
f(val)

Out[55]: 2-element Array{Int64,1}:
    5
    11

In general, these “out-of-place” functions are preferred to “in-place” functions, which modify the arguments.

In [56]: function f(x)
    return [1 2; 3 4] * x  # matrix * column vector
end

val = [1, 2]
y = similar(val)

function f!(out, x)
    out .= [1 2; 3 4] * x
end

f!(y, val)
y

Out[56]: 2-element Array{Int64,1}:
    5
    11
This demonstrates a key convention in Julia: functions which modify any of the arguments have the name ending with `!` (e.g. `push!`).

We can also see a common mistake, where instead of modifying the arguments, the name binding is swapped

```julia
In [57]: function f(x)
    return [1 2; 3 4] * x  # matrix * column vector
end
val = [1, 2]
y = similar(val)

function f!(out, x)
    out = [1 2; 3 4] * x  # MISTAKE! Should be .= or [:]
end
f!(y, val)
y

Out[57]: 2-element Array{Int64,1}:
    5674
    140029483809008
```

The frequency of making this mistake is one of the reasons to avoid in-place functions, unless proven to be necessary by benchmarking.

### 5.3.7 In-place and Immutable Types

Note that scalars are always immutable, such that

```julia
In [58]: y = [1 2]
y .= 2  # y .= y .- 2, no problem
x = 5
   # x .= 2  # Fails!
x = x .- 2  # subtle difference - creates a new value and rebinds the variable

Out[58]: 3
```

In particular, there is no way to pass any immutable into a function and have it modified

```julia
In [59]: x = 2

function f(x)
    x = 3  # MISTAKE! does not modify x, creates a new value!
end
f(x)  # cannot modify imutables in place
@show x;

x = 2
```
This is also true for other immutable types such as tuples, as well as some vector types.

In [60]:
using StaticArrays
xdynamic = [1, 2]
xstatic = @SVector [1, 2]  # turns it into a highly optimized static vector

f(x) = 2x
@show f(xdynamic)
@show f(xstatic)

# inplace version
function g(x)
    x .= 2x
    return "Success!"
end
@show xdynamic
@show g(xdynamic)
@show xdynamic;

# g(xstatic) # fails, static vectors are immutable

f(xdynamic) = [2, 4]
f(xstatic) = [2, 4]
xdynamic = [1, 2]
g(xdynamic) = "Success!"
xdynamic = [2, 4]

5.4 Operations on Arrays

5.4.1 Array Methods

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

In [61]: a = [-1, 0, 1]

@show length(a)
@show sum(a)
@show mean(a)
@show std(a)    # standard deviation
@show var(a)    # variance
@show maximum(a)
@show minimum(a)
@show extrema(a) # (minimum(a), maximum(a))

    length(a) = 3
sum(a) = 0
mean(a) = 0.0
std(a) = 1.0
var(a) = 1.0
maximum(a) = 1
minimum(a) = -1
extrema(a) = (-1, 1)
5.4. OPERATIONS ON ARRAYS

Out[61]: (-1, 1)

To sort an array

In [62]: b = sort(a, rev = true)  # returns new array, original not modified

Out[62]: 3-element Array{Int64,1}:
    1
    0
   -1

In [63]: b = sort!(a, rev = true)  # returns "modified original" array

Out[63]: 3-element Array{Int64,1}:
    1
    0
   -1

In [64]: b == a  # tests if have the same values

Out[64]: true

In [65]: b === a  # tests if arrays are identical (i.e share same memory)

Out[65]: true

5.4.2 Matrix Algebra

For two dimensional arrays, * means matrix multiplication

In [66]: a = ones(1, 2)

Out[66]: 1×2 Array{Float64,2}:
    1.0  1.0

In [67]: b = ones(2, 2)

Out[67]: 2×2 Array{Float64,2}:
    1.0  1.0
    1.0  1.0

In [68]: a * b

Out[68]: 1×2 Array{Float64,2}:
    2.0  2.0

In [69]: b * a'
To solve the linear system $AX = B$ for $X$ use $A \backslash B$

In [70]: $A = [1 \ 2; \ 2 \ 3]$

Out[70]: 2×2 Array{Int64,2}:
1 2
2 3

In [71]: $B = \text{ones}(2, 2)$

Out[71]: 2×2 Array{Float64,2}:
1.0 1.0
1.0 1.0

In [72]: $A \backslash B$

Out[72]: 2×2 Array{Float64,2}:
-1.0 -1.0
1.0 1.0

In [73]: \text{inv}(A) \ast B$

Out[73]: 2×2 Array{Float64,2}:
-1.0 -1.0
1.0 1.0

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

Multiplying two one dimensional vectors gives an error – which is reasonable since the meaning is ambiguous.

More precisely, the error is that there isn’t an implementation of $\ast$ for two one dimensional vectors.

The output explains this, and lists some other methods of $\ast$ which Julia thinks are close to what we want.

In [74]: $\text{ones}(2) \ast \text{ones}(2)$ \# does not conform, expect error

    MethodError: no method matching *(::Array{Float64,1}, ::Array{Float64,1})
    Closest candidates are:
    *(::Any, ::Any, !Matched::Any, !Matched::Any...) at operators.jl:
    529
5.4. OPERATIONS ON ARRAYS

*(!Matched::Adjoint{#s662,#s661} where #s661<:
  Union{DenseArray{T,2}, Base.ReinterpretArray{T,2,S,A} where S} where A<:
  DenseArray where N where T, DenseArray}, Base.
  ReshapedArray{T,2,A,MI} where MI<:Tuple{Vararg{Any,N} where N} where A<:
  DenseArray where N where T, SubArray{T,N,A,I,true} where I<:
  Tuple{AbstractUnitRange,Vararg{Any,N} where N}} where A<:
  DenseArray where N where T, DenseArray}, Base.
  ReshapedArray{T,2,A,MI} where MI<:Tuple{Vararg{Any,N} where N} where A<:
  DenseArray where N where T, SubArray{T,N,A,I,true} where I<:
  Tuple{AbstractUnitRange,Vararg{Any,N} where N}} where A<:
  DenseArray where N where T, DenseArray}, Base.
  ReshapedArray{T,2,A,MI} where MI<:Tuple{Vararg{Any,N} where N} where A<:
  DenseArray where N where T, SubArray{T,N,A,I,true} where I<:
  Tuple{AbstractUnitRange,Vararg{Any,N} where N}} where A<:
  DenseArray where N where T, DenseArray}, Base.
  ReshapedArray{T,2,A,MI} where MI<:Tuple{Vararg{Any,N} where N} where A<:
  DenseArray where N where T, SubArray{T,N,A,I,true} where I<:
  Tuple{AbstractUnitRange,Vararg{Any,N} where N}} where A<:
  DenseArray where N where T, DenseArray}, Base.
  ReshapedArray{T,2,A,MI} where MI<:Tuple{Vararg{Any,N} where N} where A<:
  DenseArray where N where T, SubArray{T,N,A,I,true} where I<:
  Tuple{AbstractUnitRange,Vararg{Any,N} where N}} where A<:
  DenseArray where N where T, DenseArray}, Base.
  ReshapedArray{T,2,A,MI} where MI<:Tuple{Vararg{Any,N} where N} where A<:
  DenseArray where N where T, SubArray{T,N,A,I,true} where I<:
  Tuple{AbstractUnitRange,Vararg{Any,N} where N}} where A<:
  DenseArray where N where T, DenseArray}, Base.
  ReshapedArray{T,2,A,MI} where MI<:Tuple{Vararg{Any,N} where N} where A<:
  DenseArray where N where T, SubArray{T,N,A,I,true} where I<:
  Tuple{AbstractUnitRange,Vararg{Any,N} where N}} where A<:
  DenseArray where N where T, DenseArray}, Base.
  ReshapedArray{T,2,A,MI} where MI<:Tuple{Vararg{Any,N} where N} where A<:
  DenseArray where N where T, SubArray{T,N,A,I,true} where I<:
  Tuple{AbstractUnitRange,Vararg{Any,N} where N}} where A<:
  DenseArray where N where T, DenseArray}, Base.
  ReshapedArray{T,2,A,MI} where MI<:Tuple{Vararg{Any,N} where N} where A<:
  DenseArray where N where T, SubArray{T,N,A,I,true} where I<:
  Tuple{AbstractUnitRange,Vararg{Any,N} where N}} where A<:
  DenseArray where N where T, DenseArray}, Base.
  ReshapedArray{T,2,A,MI} where MI<:Tuple{Vararg{Any,N} where N} where A<:
  DenseArray where N where T, SubArray{T,N,A,I,true} where I<:
  Tuple{AbstractUnitRange,Vararg{Any,N} where N}} where A<:
  DenseArray where N where T, DenseArray}, Base.
  ReshapedArray{T,2,A,MI} where MI<:Tuple{Vararg{Any,N} where N} where A<:
  DenseArray where N where T, SubArray{T,N,A,I,true} where I<:
  Tuple{AbstractUnitRange,Vararg{Any,N} where N}} where A<:
  DenseArray where N where T, DenseArray}, Base.
  ReshapedArray{T,2,A,MI} where MI<:Tuple{Vararg{Any,N} where N} where A<:
  DenseArray where N where T, SubArray{T,N,A,I,true} where I<:
  Tuple{AbstractUnitRange,Vararg{Any,N} where N}} where A<:
  DenseArray where N where T, DenseArray}, Base.
  ReshapedArray{T,2,A,MI} where MI<:Tuple{Vararg{Any,N} where N} where A<:
  DenseArray where N where T, SubArray{T,N,A,I,true} where I<:
  Tuple{AbstractUnitRange,Vararg{Any,N} where N}} where A<:
  DenseArray where N where T, DenseArray}, Base.
  ReshapedArray{T,2,A,MI} where MI<:Tuple{Vararg{Any,N} where N} where A<:
  DenseArray where N where T, SubArray{T,N,A,I,true} where I<:
  Tuple{AbstractUnitRange,Vararg{Any,N} where N}} where A<:
  DenseArray where N where T, DenseArray}, Base.
  ReshapedArray{T,2,A,MI} where MI<:Tuple{Vararg{Any,N} where N} where A<:
  DenseArray where N where T, SubArray{T,N,A,I,true} where I<:
  Tuple{AbstractUnitRange,Vararg{Any,N} where N}} where A<:
  DenseArray where N where T, DenseArray}, Base.
  ReshapedArray{T,2,A,MI} where MI<:Tuple{Vararg{Any,N} where N} where A<:
  DenseArray where N where T, SubArray{T,N,A,I,true} where I<:
  Tuple{AbstractUnitRange,Vararg{Any,N} where N}} where A<:
  DenseArray where N where T, DenseArray}, Base.
  ReshapedArray{T,2,A,MI} where MI<:Tuple{Vararg{Any,N} where N} where A<:
  DenseArray where N where T, SubArray{T,N,A,I,true} where I<:
  Tuple{AbstractUnitRange,Vararg{Any,N} where N}} where A<:
  DenseArray where N where T, DenseArray}, Base.
  ReshapedArray{T,2,A,MI} where MI<:Tuple{Vararg{Any,N} where N} where A<:
  DenseArray where N where T, SubArray{T,N,A,I,true} where I<:
  Tuple{AbstractUnitRange,Vararg{Any,N} where N}} where A<:
  DenseArray where N where T, DenseArray}, Base.
  ReshapedArray{T,2,A,MI} where MI<:Tuple{Vararg{Any,N} where N} where A<:
  DenseArray where N where T, SubArray{T,N,A,I,true} where I<:
  Tuple{AbstractUnitRange,Vararg{Any,N} where N}} where A<:
  DenseArray where N where T, DenseArray}, Base.
  ReshapedArray{T,2,A,MI} where MI<:Tuple{Vararg{Any,N} where N} where A<:
  DenseArray where N where T, SubArray{T,N,A,I,true} where I<:
  Tuple{AbstractUnitRange,Vararg{Any,N} where N}} where A<:
  DenseArray where N where T, DenseArray}, Base.
  ReshapedArray{T,2,A,MI} where MI<:Tuple{Vararg{Any,N} where N} where A<:
  DenseArray where N where T, SubArray{T,N,A,I,true} where I<:
  Tuple{AbstractUnitRange,Vararg{Any,N} where N}} where A<:
  DenseArray where N where T, DenseArray}, Base.
  ReshapedArray{T,2,A,MI} where MI<:Tuple{Vararg{Any,N} where N} where A<:
  DenseArray where N where T, SubArray{T,N,A,I,true} where I<:
  Tuple{AbstractUnitRange,Vararg{Any,N} where N}} where A<:
  DenseArray where N where T, DenseArray}, Base.
  ReshapedArray{T,2,A,MI} where MI<:Tuple{Vararg{Any,N} where N} where A<:
  DenseArray where N where T, SubArray{T,N,A,I,true} where I<:
  Tuple{AbstractUnitRange,Vararg{Any,N} where N}} where A<:
  DenseArray where N where T, DenseArray}, Base.
  ReshapedArray{T,2,A,MI} where MI<:Tuple{Vararg{Any,N} where N} where A<:
  DenseArray where N where T, SubArray{T,N,A,I,true} where I<:
  Tuple{AbstractUnitRange,Vararg{Any,N} where N}} where A<:
  DenseArray where N where T, DenseArray}, Base.
  ReshapedArray{T,2,A,MI} where MI<:Tuple{Vararg{Any,N} where N} where A<:
  DenseArray where N where T, SubArray{T,N,A,I,true} where I<:
  Tuple{AbstractUnitRange,Vararg{Any,N} where N}} where A<:
  DenseArray where N where T, DenseArray}, Base.
  ReshapedArray{T,2,A,MI} where MI<:Tuple{Vararg{Any,N} where N} where A<:
  DenseArray where N where T, SubArray{T,N,A,I,true} where I<:
  Tuple{AbstractUnitRange,Vararg{Any,N} where N}} where A<:
  DenseArray where N where T, DenseArray}, Base.
  ReshapedArray{T,2,A,MI} where MI<:Tuple{Vararg{Any,N} where N} where A<:
  DenseArray where N where T, SubArray{T,N,A,I,true} where I<:
  Tuple{AbstractUnitRange,Vararg{Any,N} where N}} where A<:
  DenseArray where N where T, DenseArray}, Base.
  ReshapedArray{T,2,A,MI} where MI<:Tuple{Vararg{Any,N} where N} where A<:
Instead, you could take the transpose to form a row vector

In [75]: ones(2)' * ones(2)

Out[75]: 2.0

Alternatively, for inner product in this setting use `dot()` or the unicode `\cdot<TAB>`

In [76]:@show dot(ones(2), ones(2))
  @show ones(2) ⬤ ones(2);

```
dot(ones(2), ones(2)) = 2.0
ones(2) ⬤ ones(2) = 2.0
```

Matrix multiplication using one dimensional vectors similarly follows from treating them as column vectors. Post-multiplication requires a transpose

In [77]: b = ones(2, 2)
   b * ones(2)

Out[77]: 2-element Array{Float64,1}:
   2.0
   2.0

In [78]: ones(2)' * b

Out[78]: 1x2 Adjoint{Float64,Array{Float64,1}}:
   2.0  2.0

Note that the type of the returned value in this case is not `Array{Float64,1}` but rather `Adjoint{Float64,Array{Float64,1}}`.

This is since the left multiplication by a row vector should also be a row-vector. It also hints that the types in Julia more complicated than first appears in the surface notation, as we will explore further in the introduction to types lecture.
5.4. OPERATIONS ON ARRAYS

5.4.3 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

In [79]: ones(2, 2) * ones(2, 2)  # matrix multiplication

Out[79]: 2×2 Array{Float64,2}:
    2.0  2.0
    2.0  2.0

In [80]: ones(2, 2) .* ones(2, 2)  # element by element multiplication

Out[80]: 2×2 Array{Float64,2}:
    1.0  1.0
    1.0  1.0

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

In [81]: A = -ones(2, 2)

Out[81]: 2×2 Array{Float64,2}:
    -1.0 -1.0
    -1.0 -1.0

In [82]: A.^2  # square every element

Out[82]: 2×2 Array{Float64,2}:
    1.0  1.0
    1.0  1.0

However in practice some operations are mathematically valid without broadcasting, and hence the $.$ can be omitted.

In [83]: ones(2, 2) + ones(2, 2)  # same as ones(2, 2) .+ ones(2, 2)

Out[83]: 2×2 Array{Float64,2}:
    2.0  2.0
    2.0  2.0

Scalar multiplication is similar

In [84]: A = ones(2, 2)
**CHAPTER 5. ARRAYS, TUPLES, RANGES, AND OTHER FUNDAMENTAL TYPES**

Out[84]: $2 \times 2$ Array\{Float64,2\}:

```
   1.0  1.0
   1.0  1.0
```

In [85]: $2 \times A$  # same as $2 \cdot A$

Out[85]: $2 \times 2$ Array\{Float64,2\}:

```
   2.0  2.0
   2.0  2.0
```

In fact you can omit the $\cdot$ altogether and just write $2A$.

Unlike MATLAB and other languages, scalar addition requires the .+ in order to correctly broadcast.

In [86]: $x = [1, 2]$  
$x .+ 1$  # not $x + 1$  
$x .- 1$  # not $x - 1$

Out[86]: 2-element Array\{Int64,1\}:

```
   0
   1
```

### Elementwise Comparisons

Elementwise comparisons also use the .\* style notation.

In [87]: $a = [10, 20, 30]$  

Out[87]: 3-element Array\{Int64,1\}:

```
   10
   20
   30
```

In [88]: $b = [-100, 0, 100]$  

Out[88]: 3-element Array\{Int64,1\}:

```
  -100
    0
   100
```

In [89]: $b .> a$

Out[89]: 3-element BitArray\{1\}:

```
    0
    0
    1
```

In [90]: $a .== b$
5.4. **OPERATIONS ON ARRAYS**

We can also do comparisons against scalars with parallel syntax:

```python
In [91]: b
Out[91]: 3-element Array{Int64,1}:
         -100
          0
         100

In [92]: b .> 1
Out[92]: 3-element BitArray{1}:
         0
          0
          1
```

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

```python
In [93]: a = randn(4)
Out[93]: 4-element Array{Float64,1}:
         -0.6434467928769482
         -0.35303489730240734
          0.9772679080446901
         -1.4015326160821104

In [94]: a .< 0
Out[94]: 4-element BitArray{1}:
         1
          1
          0
          1
```

```python
In [95]: a[a .< 0]
Out[95]: 3-element Array{Float64,1}:
         -0.6434467928769482
         -0.35303489730240734
         -1.4015326160821104
```

**Changing Dimensions**

The primary function for changing the dimensions of an array is `reshape()`
In [96]: a = [10, 20, 30, 40]

Out[96]: 4-element Array{Int64,1}:
    10
    20
    30
    40

In [97]: b = reshape(a, 2, 2)

Out[97]: 2×2 Array{Int64,2}:
    10  30
    20  40

In [98]: b

Out[98]: 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.

In [99]: b[1, 1] = 100  # continuing the previous example

Out[99]: 100

In [100]: b

Out[100]: 2×2 Array{Int64,2}:
    100  30
    20  40

In [101]: a

Out[101]: 4-element Array{Int64,1}:
    100
    20
    30
    40

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

In [102]: a = [1 2 3 4]  # two dimensional

Out[102]: 1×4 Array{Int64,2}:
    1  2  3  4

In [103]: dropdims(a, dims = 1)
5.4. OPERATIONS ON ARRAYS

Out[103]: 4-element Array{Int64,1}:
  1
  2
  3
  4

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

5.4.4 Broadcasting Functions

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

In [104]: \texttt{log(1.0)}

Out[104]: 0.0

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

In [105]: \texttt{log.(1:4)}

Out[105]: 4-element Array{Float64,1}:
 0.0
 0.6931471805599453
 1.0986122886681098
 1.3862943611198906

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

In [106]: \texttt{[ log(x) for x in 1:4 ]}

Out[106]: 4-element Array{Float64,1}:
 0.0
 0.6931471805599453
 1.0986122886681098
 1.3862943611198906

Nonetheless the syntax is convenient.

5.4.5 Linear Algebra

(See linear algebra documentation)

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

In [107]: \texttt{A = [1 2; 3 4]}

Out[107]: 2×2 Array{Int64,2}:
 1 2
 3 4
In [108]: det(A)
Out[108]: -2.0

In [109]: tr(A)
Out[109]: 5

In [110]: eigvals(A)
Out[110]: 2-element Array{Float64,1}:
          -0.3722813232690143
          5.372281323269014

In [111]: rank(A)
Out[111]: 2

5.5 Ranges

As with many other types, a Range can act as a vector.

In [112]: a = 10:12          # a range, equivalent to 10:1:12
    # can convert, but shouldn’t
    b = Diagonal([1.0, 2.0, 3.0])
    b * a -: [1.0; 2.0; 3.0]

    Vector(a) = [10, 11, 12]

Out[112]: 3-element Array{Float64,1}:
          9.0
          20.0
          33.0

Ranges can also be created with floating point numbers using the same notation.

In [113]: a = 0.0:0.1:1.0   # 0.0, 0.1, 0.2, ... 1.0

Out[113]: 0.0:0.1:1.0

But care should be taken if the terminal node is not a multiple of the set sizes.

In [114]: maxval = 1.0
    minval = 0.0
    stepsize = 0.15
    a = minval:stepsize:maxval  # 0.0, 0.15, 0.3, ...
    maximum(a) == maxval
To evenly space points where the maximum value is important, i.e., \texttt{linspace} in other languages

\begin{verbatim}
maxval = 1.0
minval = 0.0
numpoints = 10
a = range(minval, maxval, length=numpoints)
# or range(minval, stop=maxval, length=numpoints)

maximum(a) == maxval
\end{verbatim}

We were introduced to tuples earlier, which provide high-performance immutable sets of distinct types.

\begin{verbatim}
t = (1.0, "test")
t[1]  # access by index
a, b = t  # unpack
# t[1] = 3.0  # would fail as tuples are immutable
println("a = $a and b = $b")

a = 1.0 and b = test
\end{verbatim}

As well as named tuples, which extend tuples with names for each argument.

\begin{verbatim}
t = (val1 = 1.0, val2 = "test")
t.val1  # access by index
# a, b = t  # bad style, better to unpack by name with @unpack
println("val1 = $(t.val1) and val1 = $(t.val1)")  # access by name

val1 = 1.0 and val1 = 1.0
\end{verbatim}

While immutable, it is possible to manipulate tuples and generate new ones

\begin{verbatim}
t2 = (val3 = 4, val4 = "test!!")
t3 = merge(t, t2)  # new tuple
\end{verbatim}

Named tuples are a convenient and high-performance way to manage and unpack sets of parameters.
In [119]: \texttt{function \ } f(\texttt{parameters})\
\texttt{\quad \alpha, \ \beta = \texttt{parameters.}\alpha, \ \texttt{parameters.}\beta \ # \ \texttt{poor \ style, \ error \ prone \ if}}\
\texttt{\quad \texttt{-}adding \ \texttt{parameters}}\
\texttt{\quad \texttt{return} \ \alpha + \beta}\
\texttt{\quad end}\
\texttt{parameters = (\alpha = 0.1, \ \beta = 0.2)}\
\texttt{f(parameters)}\

\texttt{Out[119]: 0.30000000000000004}

This functionality is aided by the \texttt{Parameters.jl} package and the \texttt{@unpack} macro

In [120]: \texttt{using Parameters}\
\texttt{function f(\texttt{parameters})}\
\texttt{\quad \texttt{@unpack} \ \alpha, \ \beta = \texttt{parameters} \ # \ \texttt{good \ style, \ less \ sensitive \ to \ errors}}\
\texttt{\quad \texttt{return} \ \alpha + \beta}\
\texttt{\quad end}\
\texttt{parameters = (\alpha = 0.1, \ \beta = 0.2)}\
\texttt{f(parameters)}\

\texttt{Out[120]: 0.30000000000000004}

In order to manage default values, use the \texttt{@with_kw} macro

In [121]: \texttt{using Parameters}\
\texttt{\texttt{paramgen} = \texttt{\texttt{@with_kw}} (\alpha = 0.1, \ \beta = 0.2) \ # \ \texttt{create \ named \ tuples \ with \ defaults}}\
\texttt{\quad \texttt{# \ \texttt{creates \ named \ tuples, \ replacing \ defaults}}\
\texttt{\quad \texttt{@show \ \texttt{paramgen}()} \ # \ \texttt{calling \ without \ arguments \ gives \ all \ defaults}}\
\texttt{\quad \texttt{@show \ \texttt{paramgen}(\alpha = 0.2)}}\
\texttt{\quad \texttt{@show \ \texttt{paramgen}(\alpha = 0.2, \ \beta = 0.5));}\

\texttt{\texttt{paramgen}()} = (\alpha = 0.1, \ \beta = 0.2)\
\texttt{\texttt{paramgen}(\alpha = 0.2)} = (\alpha = 0.2, \ \beta = 0.2)\
\texttt{\texttt{paramgen}(\alpha = 0.2, \ \beta = 0.5)} = (\alpha = 0.2, \ \beta = 0.5)

An alternative approach, defining a new type using \texttt{struct} tends to be more prone to accidental misuse, and leads to a great deal of boilerplate code.

For that, and other reasons of generality, we will use named tuples for collections of parameters where possible.

### 5.7 Nothing, Missing, and Unions

Sometimes a variable, return type from a function, or value in an array needs to represent the absence of a value rather than a particular value.

There are two distinct use cases for this
1. **nothing** (“software engineers null”): used where no value makes sense in a particular context due to a failure in the code, a function parameter not passed in, etc.

2. **missing** (“data scientists null”): used when a value would make conceptual sense, but it isn’t available.

### 5.7.1 Nothing and Basic Error Handling

The value `nothing` is a single value of type `Nothing`.

```julia
In [122]: typeof(nothing)
```

**Out[122]:** Nothing

An example of a reasonable use of `nothing` is if you need to have a variable defined in an outer scope, which may or may not be set in an inner one.

```julia
In [123]: function f(y)
    x = nothing
    if y > 0.0
        # calculations to set `x`
        x = y
    end
    # later, can check `x`
    if isnothing(x)
        println("x was not set")
    else
        println("x = \\
    end
    x
end

@show f(1.0)
@show f(-1.0);
```

```julia
x = 1.0
f(1.0) = 1.0
x was not set
f(-1.0) = nothing
```

While in general you want to keep a variable name bound to a single type in Julia, this is a notable exception.

Similarly, if needed, you can return a `nothing` from a function to indicate that it did not calculate as expected.

```julia
In [124]: function f(x)
    if x > 0.0
        return sqrt(x)
    else
        return nothing
```
end
end
x1 = 1.0
x2 = -1.0
y1 = f(x1)
y2 = f(x2)

# check results with isnothing
if isnothing(y1)
    println("f(x) successful")
else
    println("f(x) failed");
end

f(-1.0) failed

As an aside, an equivalent way to write the above function is to use the ternary operator,
which gives a compact if/then/else structure

In [125]: function f(x)
    x > 0.0 ? sqrt(x) : nothing # the "a ? b : c" pattern is the ternary
end

f(1.0)

Out[125]: 1.0

We will sometimes use this form when it makes the code more clear (and it will occasionally
make the code higher performance).

Regardless of how $f(x)$ is written, the return type is an example of a union, where the result
could be one of an explicit set of types.

In this particular case, the compiler would deduce that the type would be a
Union{Nothing, Float64} – that is, it returns either a floating point or a nothing.

You will see this type directly if you use an array containing both types

In [126]: x = [1.0, nothing]

Out[126]: 2-element Array{Union{Nothing, Float64},1}:
    1.0
    nothing

When considering error handling, whether you want a function to return nothing or simply
fail depends on whether the code calling $f(x)$ is carefully checking the results.

For example, if you were calling on an array of parameters where a priori you were not sure
which ones will succeed, then

In [127]: x = [0.1, -1.0, 2.0, -2.0]
y = f.(x)

    # presumably check `y`
On the other hand, if the parameter passed is invalid and you would prefer not to handle a graceful failure, then using an assertion is more appropriate.

```
In [128]:
   def f(x):
       @assert x > 0.0
       return sqrt(x)

f(1.0)
```

```
Out[128]: 1.0
```

Finally, `nothing` is a good way to indicate an optional parameter in a function.

```
In [129]:
   def f(x; z = nothing):
       if isnothing(z):
           println("No z given with $x")
       else:
           println("z = $z given with $x")
   end

f(1.0)
f(1.0, z=3.0)

```

```
No z given with 1.0
z = 3.0 given with 1.0
```

An alternative to `nothing`, which can be useful and sometimes higher performance, is to use `NaN` to signal that a value is invalid returning from a function.

```
In [130]:
   def f(x):
       if x > 0.0
           return x
       else:
           return NaN
   end

f(0.1)
f(-1.0)

@show typeof(f(-1.0))
@show f(-1.0) == NaN  # note, this fails!
@show isnan(f(-1.0))  # check with this
```
typeof(f(-1.0)) = Float64
f(-1.0) == NaN = false
isnan(f(-1.0)) = true

Out[130]: true

Note that in this case, the return type is Float64 regardless of the input for Float64 input.

Keep in mind, though, that this only works if the return type of a function is Float64.

5.7.2 Exceptions

(See exceptions documentation)

While returning a nothing can be a good way to deal with functions which may or may not return values, a more robust error handling method is to use exceptions.

Unless you are writing a package, you will rarely want to define and throw your own exceptions, but will need to deal with them from other libraries.

The key distinction for when to use an exceptions vs. return a nothing is whether an error is unexpected rather than a normal path of execution.

An example of an exception is a DomainError, which signifies that a value passed to a function is invalid.

In [131]: # throws exception, turned off to prevent breaking notebook
   # sqrt(-1.0)
   # to see the error
   try sqrt(-1.0); catch err; err end  # catches the exception and prints it

Out[131]: DomainError(-1.0, "sqrt will only return a complex result if called with a complex argument. Try sqrt(Complex(x)).")

Another example you will see is when the compiler cannot convert between types.

In [132]: # throws exception, turned off to prevent breaking notebook
   # convert(Int64, 3.12)
   # to see the error
   try convert(Int64, 3.12); catch err; err end  # catches the exception and prints it

Out[132]: InexactError(:Int64, Int64, 3.12)

If these exceptions are generated from unexpected cases in your code, it may be appropriate simply let them occur and ensure you can read the error.

Occasionally you will want to catch these errors and try to recover, as we did above in the try block.
5.7. NOTHING, MISSING, AND UNIONS

In [133]: function f(x)
    try
        sqrt(x)
    catch err
        # enters if exception thrown
        sqrt(complex(x, 0)) # convert to complex number
    end
end

f(0.0)
f(-1.0)

Out[133]: 0.0 + 1.0im

5.7.3 Missing

(see “missing” documentation)

The value missing of type Missing is used to represent missing value in a statistical sense.

For example, if you loaded data from a panel, and gaps existed

In [134]: x = [3.0, missing, 5.0, missing, missing]

Out[134]: 5-element Array{Union{Missing, Float64},1}:
            3.0
            missing
            5.0
            missing
            missing

A key feature of missing is that it propagates through other function calls - unlike nothing

In [135]: f(x) = x^2

   @show missing + 1.0
   @show missing * 2
   @show missing * "test"
   @show f(missing);       # even user-defined functions
   @show mean(x);

    missing + 1.0 = missing
    missing * 2 = missing
    missing * "test" = missing
    f(missing) = missing
    mean(x) = missing

The purpose of this is to ensure that failures do not silently fail and provide meaningless numerical results.

This even applies for the comparison of values, which
In [136]: x = missing

   @show x == missing
   @show x === missing  # an exception
    @show ismissing(x);

    x == missing = missing
    x === missing = true
    ismissing(x) = true

Where ismissing is the canonical way to test the value.

In the case where you would like to calculate a value without the missing values, you can use skipmissing.

In [137]: x = [1.0, missing, 2.0, missing, missing, 5.0]

    @show mean(x)
    @show mean(skipmissing(x))
    @show coalesce.(x, 0.0);  # replace missing with 0.0;

mean(x) = missing
mean(skipmissing(x)) = 2.6666666666666665
coalesce.(x, 0.0) = [1.0, 0.0, 2.0, 0.0, 0.0, 5.0]

As missing is similar to R’s NA type, we will see more of missing when we cover DataFrames.

5.8 Exercises

5.8.1 Exercise 1

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

If you aren’t familiar with all the terminology don’t be concerned – you can skim read the background discussion and focus purely on the matrix exercise.

With that said, consider the stochastic difference equation

\[ X_{t+1} = AX_t + b + \Sigma W_{t+1} \]  

(1)

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) that \( \{S_t\} \) obeys

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

(2)

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 (2) given \( A \) and \( \Sigma \).

To test your solution, observe that the limit \( S \) is a solution to the matrix equation

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

(3)

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

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

5.8.2 Exercise 2

Take a stochastic process for \( \{y_t\}_{t=0}^T \)

\[
y_{t+1} = \gamma + \theta y_t + \sigma w_{t+1}
\]

where

- \( w_{t+1} \) is distributed \( \text{Normal}(0,1) \)
- \( \gamma = 1, \sigma = 1, y_0 = 0 \)
- \( \theta \in \Theta \equiv \{0.8, 0.9, 0.98\} \)

Given these parameters

- Simulate a single \( y_t \) series for each \( \theta \in \Theta \) for \( T = 150 \). Feel free to experiment with different \( T \).
- Overlay plots of the rolling mean of the process for each \( \theta \in \Theta \), i.e. for each \( 1 \leq \tau \leq T \) plot

\[
\frac{1}{\tau} \sum_{t=1}^{\tau} y_T
\]

- Simulate \( N = 200 \) paths of the stochastic process above to the \( T \), for each \( \theta \in \Theta \), where we refer to an element of a particular simulation as \( y_t^\theta \).
Overlay plots a histogram of the stationary distribution of the final $y_n^T$ for each $\theta \in \Theta$. Hint: pass `alpha` to a plot to make it transparent (e.g. `histogram(vals, alpha = 0.5)`) or use `stephist(vals)` to show just the step function for the histogram.

Numerically find the mean and variance of this as an ensemble average, i.e. $\sum_{n=1}^{N} \frac{y_n^T}{N}$ and $\sum_{n=1}^{N} \frac{(y_n^T)^2}{N} - \left( \sum_{n=1}^{N} \frac{y_n^T}{N} \right)^2$.

Later, we will interpret some of these in this lecture.

### 5.8.3 Exercise 3

Let the data generating process for a variable be

$$y = ax_1 + bx_1^2 + cx_2 + d + \sigma w$$

where $y, x_1, x_2$ are scalar observables, $a, b, c, d$ are parameters to estimate, and $w$ are iid normal with mean 0 and variance 1.

First, let’s simulate data we can use to estimate the parameters

- Draw $N = 50$ values for $x_1, x_2$ from iid normal distributions.

Then, simulate with different $w$ * Draw a $w$ vector for the $N$ values and then $y$ from this simulated data if the parameters were $a = 0.1, b = 0.2, c = 0.5, d = 1.0, \sigma = 0.1$. * Repeat that so you have $M = 20$ different simulations of the $y$ for the $N$ values.

Finally, calculate order least squares manually (i.e., put the observables into matrices and vectors, and directly use the equations for OLS rather than a package).

- For each of the $M = 20$ simulations, calculate the OLS estimates for $a, b, c, d, \sigma$.
- Plot a histogram of these estimates for each variable.

### 5.8.4 Exercise 4

Redo Exercise 1 using the `fixedpoint` function from `NLsolve` this lecture.

Compare the number of iterations of the NLsolve’s Anderson Acceleration to the handcoded iteration used in Exercise 1.

Hint: Convert the matrix to a vector to use `fixedpoint`. e.g. $A = [1 2; 3 4]$ then $x = \text{reshape}(A, 4)$ turns it into a vector. To reverse, $\text{reshape}(x, 2, 2)$.

### 5.9 Solutions

#### 5.9.1 Exercise 1

Here’s the iterative approach

In [138]: `function` compute_asymptotic_var(A, $\Sigma$;

\[
S\Theta = \Sigma \ast \Sigma',
\]

\[
tolerance = 1e-6,
\]

\[
\text{maxiter} = 500
\]

\[
V = \Sigma \ast \Sigma'
\]
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 += 1
end
return S
end

Out[138]: compute_asymptotic_var (generic function with 1 method)

In [139]: A = [0.8 -0.2; -0.1 0.7]
Σ = [0.5 0.4; 0.4 0.6]

Out[139]: 2×2 Array{Float64,2}:
0.5   0.4
0.4   0.6

Note that all eigenvalues of A lie inside the unit disc.

In [140]: maximum(abs, eigvals(A))

Out[140]: 0.9

Let’s compute the asymptotic variance

In [141]: our_solution = compute_asymptotic_var(A, Σ)

Out[141]: 2×2 Array{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.

In [142]: using QuantEcon

In [143]: norm(our_solution - solve_discrete_lyapunov(A, Σ * Σ'))

Out[143]: 3.883245447999784e-6
Chapter 6

Introduction to Types and Generic Programming

6.1 Contents

- Overview 6.2
- Finding and Interpreting Types 6.3
- The Type Hierarchy 6.4
- Deducing and Declaring Types 6.5
- Creating New Types 6.6
- Introduction to Multiple Dispatch 6.7
- Exercises 6.8

6.2 Overview

In Julia, arrays and tuples are the most important data type for working with numerical data.

In this lecture we give more details on

- declaring types
- abstract types
- motivation for generic programming
- multiple dispatch
- building user-defined types

6.2.1 Setup

In [1]: using InstantiateFromURL
   # optionally add arguments to force installation: instantiate = true,
   ϸprecompile = true
   github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

In [2]: using LinearAlgebra, Statistics

111
6.3 Finding and Interpreting Types

6.3.1 Finding The Type

As we have seen in the previous lectures, in Julia all values have a type, which can be queried using the `typeof` function

In [3]: `@show typeof(1)`
   `@show typeof(1.0);`

    typeof(1) = Int64
    typeof(1.0) = Float64

The hard-coded values 1 and 1.0 are called literals in a programming language, and the compiler deduces their types (Int64 and Float64 respectively in the example above).

You can also query the type of a value

In [4]: `x = 1`
   `typeof(x)`

Out[4]: Int64

The name x binds to the value 1, created as a literal.

6.3.2 Parametric Types

(See parametric types documentation).

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

In [5]: `@show typeof(1.0 + 1im)`
   `@show typeof(ones(2, 2));`

    typeof(1.0 + 1im) = Complex{Float64}
    typeof(ones(2, 2)) = Array{Float64,2}

We will learn more details about generic programming later, but the key is to interpret the curly brackets as swappable parameters for a given type.

For example, Array{Float64, 2} can be read as

1. **Array** is a parametric type representing a dense array, where the first parameter is the type stored, and the second is the number of dimensions.

2. **Float64** is a concrete type declaring that the data stored will be a particular size of floating point.

3. **2** is the number of dimensions of that array.
A concrete type is one where values can be created by the compiler (equivalently, one which can be the result of `typeof(x)` for some object `x`).

Values of a parametric type cannot be concretely constructed unless all of the parameters are given (themselves with concrete types).

In the case of `Complex{Float64}`

1. `Complex` is an abstract complex number type.
2. `Float64` is a concrete type declaring what the type of the real and imaginary parts of the value should store.

Another type to consider is the `Tuple` and `NamedTuple`

In [6]: `x = (1, 2.0, "test")`

```julia
@show typeof(x)
```

```julia
typeof(x) = Tuple{Int64,Float64,String}
```

Out[6]: `Tuple{Int64,Float64,String}`

In this case, `Tuple` is the parametric type, and the three parameters are a list of the types of each value.

For a named tuple

In [7]: `x = (a = 1, b = 2.0, c = "test")`

```julia
@show typeof(x)
```

```julia
typeof(x) = NamedTuple{(:a, :b, :c),Tuple{Int64,Float64,String}}
```

Out[7]: `NamedTuple{(:a, :b, :c),Tuple{Int64,Float64,String}}`

The parametric `NamedTuple` type contains two parameters: first a list of names for each field of the tuple, and second the underlying `Tuple` type to store the values.

Anytime a value is prefixed by a colon, as in the `:a` above, the type is `Symbol` – a special kind of string used by the compiler.

In [8]: `typeof(:a)`

Out[8]: `Symbol`

**Remark:** Note that, by convention, type names use CamelCase – `Array`, `AbstractArray`, etc.
6.3.3 Variables, Types, and Values

Since variables and functions are lower case by convention, this can be used to easily identify types when reading code and output.

After assigning a variable name to a value, we can query the type of the value via the name.

In [9]: x = 42
   @show typeof(x);

    typeof(x) = Int64

Thus, x is just a symbol bound to a value of type Int64.

We can rebind the symbol x to any other value, of the same type or otherwise.

In [10]: x = 42.0

Out[10]: 42.0

Now x “points to” another value, of type Float64

In [11]: typeof(x)

Out[11]: Float64

However, beyond a few notable exceptions (e.g. nothing used for error handling), changing types is usually a symptom of poorly organized code, and makes type inference more difficult for the compiler.

6.4 The Type Hierarchy

Let’s discuss how types are organized.

6.4.1 Abstract vs Concrete Types

(See abstract types documentation)

Up to this point, most of the types we have worked with (e.g., Float64, Int64) are examples of concrete types.

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

We will now examine abstract types that cannot be instantiated (e.g., Real, AbstractFloat).

For example, while you will never have a Real number directly in memory, the abstract types help us organize and work with related concrete types.
6.4. THE TYPE HIERARCHY

6.4.2 Subtypes and Supertypes

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

In the Julia language specification, the types form a hierarchy.

You can check if a type is a subtype of another with the `<:` operator.

```
In [12]: @show Float64 <: Real
   @show Int64 <: Real
   @show Complex{Float64} <: Real
   @show Array <: Real;

   Float64 <: Real = true
   Int64 <: Real = true
   Complex{Float64} <: Real = false
   Array <: Real = false
```

In the above, both `Float64` and `Int64` are subtypes of `Real`, whereas the `Complex` numbers are not.

They are, however, all subtypes of `Number`

```
In [13]: @show Real <: Number
   @show Float64 <: Number
   @show Int64 <: Number
   @show Complex{Float64} <: Number;

   Real <: Number = true
   Float64 <: Number = true
   Int64 <: Number = true
   Complex{Float64} <: Number = true
```

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

```
In [14]: Number <: Any
```

```
Out[14]: 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.

There are some further functions to help you explore the type hierarchy, such as `show_supertypes` which walks up the tree of types to `Any` for a given type.

```
In [15]: using Base; show_supertypes # import the function from the `Base` package
   show_supertypes(Int64)
```
CHAPTER 6. INTRODUCTION TO TYPES AND GENERIC PROGRAMMING

Int64 <: Signed <: Integer <: Real <: Number <: Any

And the subtypes which gives a list of the available subtypes for any packages or code currently loaded

In [16]: @show subtypes(Real)
   @show subtypes(AbstractFloat);

   subtypes(Real) = Any[AbstractFloat, AbstractIrrational, Integer, Rational]
   subtypes(AbstractFloat) = Any[BigFloat, Float16, Float32, Float64]

6.5 Deducing and Declaring Types

We will discuss this in detail in generic programming, but much of Julia’s performance gains and generality of notation comes from its type system.

For example

In [17]: x1 = [1, 2, 3]
   x2 = [1.0, 2.0, 3.0]

   @show typeof(x1)
   @show typeof(x2)

   typeof(x1) = Array{Int64,1}
   typeof(x2) = Array{Float64,1}

Out[17]: Array{Float64,1}

These return Array{Int64,1} and Array{Float64,1} respectively, which the compiler is able to infer from the right hand side of the expressions.

Given the information on the type, the compiler can work through the sequence of expressions to infer other types.

In [18]: f(y) = 2y # define some function

   x = [1, 2, 3]
   z = f(x) # call with an integer array - compiler deduces type

Out[18]: 3-element Array{Int64,1}:
   2
   4
   6
6.5. Good Practices for Functions and Variable Types

In order to keep many of the benefits of Julia, you will sometimes want to ensure the compiler can always deduce a single type from any function or expression.

An example of bad practice is to use an array to hold unrelated types

In [19]: x = [1.0, "test", 1]  # typically poor style

Out[19]: 3-element Array{Any,1}:
  1.0
  "test"
  1

The type of this array is Array{Any,1}, where Any means the compiler has determined that any valid Julia type can be added to the array.

While occasionally useful, this is to be avoided whenever possible in performance sensitive code.

The other place this can come up is in the declaration of functions.

As an example, consider a function which returns different types depending on the arguments.

In [20]: function f(x)
    if x > 0
        return 1.0
    else
        return 0  # probably meant `0.0`
    end
end

@show f(1)
@show f(-1);

f(1) = 1.0
f(-1) = 0

The issue here is relatively subtle: 1.0 is a floating point, while 0 is an integer.

Consequently, given the type of x, the compiler cannot in general determine what type the function will return.

This issue, called type stability, is at the heart of most Julia performance considerations.

Luckily, trying to ensure that functions return the same types is also generally consistent with simple, clear code.

6.5.2 Manually Declaring Function and Variable Types

(See type declarations documentation)

You will notice that in the lecture notes we have never directly declared any types.

This is intentional both for exposition and as a best practice for using packages (as opposed to writing new packages, where declaring these types is very important).
It is also in contrast to some of the sample code you will see in other Julia sources, which you will need to be able to read.

To give an example of the declaration of types, the following are equivalent

In [21]:

```julia
function f(x, A)
    b = [5.0, 6.0]
    return A * x .+ b
end

val = f([0.1, 2.0], [1.0 2.0; 3.0 4.0])
```

Out[21]: 2-element Array{Float64,1}:

9.1
14.3

In [22]:

```julia
function f2(x::Vector{Float64}, A::Matrix{Float64})::Vector{Float64}
    # argument and return types
    b::Vector{Float64} = [5.0, 6.0]
    return A * x .+ b
end

val = f2([0.1; 2.0], [1.0 2.0; 3.0 4.0])
```

Out[22]: 2-element Array{Float64,1}:

9.1
14.3

While declaring the types may be verbose, would it ever generate faster code?

The answer is almost never.

Furthermore, it can lead to confusion and inefficiencies since many things that behave like vectors and matrices are not `Matrix{Float64}` and `Vector{Float64}`.

Here, the first line works and the second line fails

In [23]:

```julia
@show f([0.1; 2.0], [1 2; 3 4])
@show f([0.1; 2.0], Diagonal([1.0, 2.0]))

# f2([0.1; 2.0], [1 2; 3 4]) # not a `Float64`
# f2([0.1; 2.0], Diagonal([1.0, 2.0])) # not a `Matrix{Float64}`

f([0.1; 2.0], [1 2; 3 4]) = [9.1, 14.3]
f([0.1; 2.0], Diagonal([1.0, 2.0])) = [5.1, 10.0]
```

Out[23]: 2-element Array{Float64,1}:

5.1
10.0

### 6.6 Creating New Types

(See type declarations documentation)
Up until now, we have used `NamedTuple` to collect sets of parameters for our models and examples.

These are useful for maintaining values for model parameters, but you will eventually need to be able to use code that creates its own types.

### 6.6.1 Syntax for Creating Concrete Types

(See composite types documentation)

While other sorts of types exist, we almost always use the `struct` keyword, which is for creation of composite data types

- “Composite” refers to the fact that the data types in question can be used as collection of named fields.
- 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 has fields named `a`, `b`, `c`, are not typed

```python
In [24]: struct FooNotTyped  # immutable by default, use `mutable struct` otherwise
    a  # BAD! not typed
    b
    c
end
```

And another where the types of the fields are chosen

```python
In [25]: struct Foo
    a::Float64
    b::Int64
    c::Vector{Float64}
end
```

In either case, the compiler generates a function to create new values of the data type, called a “constructor”.

It has the same name as the data type but uses function call notion

```python
In [26]: foo_nt = FooNotTyped(2.0, 3, [1.0, 2.0, 3.0])  # new `FooNotTyped`
    foo = Foo(2.0, 3, [1.0, 2.0, 3.0])  # creates a new `Foo`
    @show typeof(foo)
    @show foo.a  # get the value for a field
    @show foo.b
    @show foo.c;
    # foo.a = 2.0  # fails since it is immutable

    typeof(foo) = Foo
    foo.a = 2.0
    foo.b = 3
    foo.c = [1.0, 2.0, 3.0]
```
You will notice two differences above for the creation of a `struct` compared to our use of `NamedTuple`.

- Types are declared for the fields, rather than inferred by the compiler.
- The construction of a new instance has no named parameters to prevent accidental misuse if the wrong order is chosen.

### 6.6.2 Issues with Type Declarations

Was it necessary to manually declare the types `a::Float64` in the above struct?

The answer, in practice, is usually yes.

Without a declaration of the type, the compiler is unable to generate efficient code, and the use of a `struct` declared without types could drop performance by orders of magnitude.

Moreover, it is very easy to use the wrong type, or unnecessarily constrain the types.

The first example, which is usually just as low-performance as no declaration of types at all, is to accidentally declare it with an abstract type

```plaintext
In [27]: struct Foo2
    a::Float64
    b::Integer  # BAD! Not a concrete type
    c::Vector<Real>  # BAD! Not a concrete type
end
```

The second issue is that by choosing a type (as in the `Foo` above), you may be unnecessarily constraining what is allowed

```plaintext
In [28]: f(x) = x.a + x.b + sum(x.c)  # use the type
    a = 2.0
    b = 3
    c = [1.0, 2.0, 3.0]
    foo = Foo(a, b, c)
   @show f(foo)  # call with the foo, no problem

    # some other typed for the values
    a = 2  # not a floating point but `f()` would work
    b = 3
    c = [1.0, 2.0, 3.0]'  # transpose is not a `Vector` but `f()` would work
    # foo = Foo(a, b, c)  # fails to compile

    # works with `NotTyped` version, but low performance
    foo_nt = FooNotTyped(a, b, c)
   @show f(foo_nt);
```

```plaintext
f(foo) = 11.0
f(foo_nt) = 11.0
```

### 6.6.3 Declaring Parametric Types (Advanced)

(See `type parametric types documentation`
Motivated by the above, we can create a type which can adapt to holding fields of different types.

**In [29]:**

```julia
struct Foo3{T1, T2, T3}
    a::T1  # could be any type
    b::T2
    c::T3
end

# works fine
a = 2
b = 3
c = [1.0, 2.0, 3.0]'
foo = Foo3(a, b, c)
@show typeof(foo)
f(foo)
```

```plaintext
typeof(foo) = Foo3{Int64,Int64,Adjoint{Float64,Array{Float64,1}}}
```

**Out[29]:** 11.0

Of course, this is probably too flexible, and the f function might not work on an arbitrary set of a, b, c.

You could constrain the types based on the abstract parent type using the <: operator.

**In [30]:**

```julia
struct Foo4{T1 <: Real, T2 <: Real, T3 <: AbstractVecOrMat{<:Real}}
    a::T1
    b::T2
    c::T3  # should check dimensions as well
end
foo = Foo4(a, b, c)  # no problem, and high performance
@show typeof(foo)
f(foo)
```

```plaintext
typeof(foo) = Foo4{Int64,Int64,Adjoint{Float64,Array{Float64,1}}}
```

**Out[30]:** 11.0

This ensures that

- a and b are a subtype of Real, and + in the definition of f works
- c is a one dimensional abstract array of Real values

The code works, and is equivalent in performance to a NamedTuple, but is more verbose and error prone.

### 6.6.4 Keyword Argument Constructors (Advanced)

There is no way to avoid learning parametric types to achieve high performance code.

However, the other issue where constructor arguments are error-prone, can be remedied with the Parameters.jl library.
In [31]: using Parameters

@with_kw struct Foo5
    a::Float64 = 2.0  # adds default value
    b::Int64
    c::Vector{Float64}
end

foo = Foo5(a = 0.1, b = 2, c = [1.0, 2.0, 3.0])
foo2 = Foo5(c = [1.0, 2.0, 3.0], b = 2)  # rearrange order, uses default

@show foo
@show foo2

function f(x)
    @unpack a, b, c = x  # can use `@unpack` on any struct
    return a + b + sum(c)
end

f(foo)

foo = Foo5
  a: Float64 0.1
  b: Int64 2
  c: Array{Float64}((3,)) [1.0, 2.0, 3.0]

foo2 = Foo5
  a: Float64 2.0
  b: Int64 2
  c: Array{Float64}((3,)) [1.0, 2.0, 3.0]

Out[31]: 8.1

6.6.5 Tips and Tricks for Writing Generic Functions

As discussed in the previous sections, there is major advantage to never declaring a type unless it is absolutely necessary.

The main place where it is necessary is designing code around multiple dispatch.

If you are careful to write code that doesn’t unnecessarily assume types, you will both achieve higher performance and allow seamless use of a number of powerful libraries such as automatic differentiation, static arrays, GPUs, interval arithmetic and root finding, arbitrary precision numbers, and many more packages – including ones that have not even been written yet.

A few simple programming patterns ensure that this is possible

- Do not declare types when declaring variables or functions unless necessary.

In [32]: # BAD
x = [5.0, 6.0, 2.1]

function g(x::Array{Float64, 1})  # not generic!
    y = zeros(length(x))  # not generic, hidden float!

```julia
# not generic, hidden float!
q = ones(length(x))
y .+ z .+ x .+ q
return y
end

g(x)

# GOOD
function g2(x)  # or `x::AbstractVector`
y = similar(x)
z = I
q = ones(eltype(x), length(x))  # or `fill(one(x), length(x))`
y .+ z .+ x .+ q
return y
end

g2(x)
```

Out[32]: 3-element Array{Float64,1}:
6.0
7.0
3.1

- Preallocate related vectors with similar where possible, and use eltype or typeof. This is important when using Multiple Dispatch given the different input types the function can call

In [33]: function g(x)
y = similar(x)
for i in eachindex(x)
y[i] = x[i]^2  # could broadcast
end
return y
end

g([BigInt(1), BigInt(2)])
```

Out[33]: 2-element Array{BigInt,1}:
1
4

- Use typeof or eltype to declare a type

In [34]: @show typeof([1.0, 2.0, 3.0])
@show eltype([1.0, 2.0, 3.0]);

    typeof([1.0, 2.0, 3.0]) = Array{Float64,1}
eltype([1.0, 2.0, 3.0]) = Float64

- Beware of hidden floating points

In [35]: @show typeof(ones(3))
@show typeof(ones(Int64, 3))
@show typeof(zeros(3))
@show typeof(zeros(Int64, 3));
Use `one` and `zero` to write generic code

In [36]: @show typeof(1)
@show typeof(1.0)
@show typeof(BigFloat(1.0))
@show typeof(one(BigFloat))  # gets multiplicative identity, passing in

```
typeof(1) = Int64
typeof(1.0) = Float64
typeof(BigFloat(1.0)) = BigFloat
typeof(one(BigFloat)) = BigFloat
```

This last example is a subtle, because of something called **type promotion**

- Assume reasonable type promotion exists for numeric types

In [37]: # ACCEPTABLE
```julia
function g(x::AbstractFloat)
    return x + 1.0  # assumes `1.0` can be converted to something compatible with `typeof(x)`
end
```

```
x = BigFloat(1.0)

@show typeof(g(x));  # this has "promoted" the `1.0` to a `BigFloat`

typeof(g(x)) = BigFloat
```

But sometimes assuming promotion is not enough

In [38]: # BAD
```julia
function g2(x::AbstractFloat)
    if x > 0.0  # can't efficiently call with `x::Integer`
        return x + 1.0  # OK - assumes you can promote `Float64` to
    `AbstractFloat`
    otherwise
```
6.6. 

```julia
return 0  # BAD! Returns a `Int64`
end
end

x = BigFloat(1.0)
x2 = BigFloat(-1.0)

@show typeof(g2(x))
@show typeof(g2(x2))  # type unstable

# GOOD
function g3(x)  #
    if x > zero(x)  # any type with an additive identity
        return x + one(x)  # more general but less important of a change
    else
        return zero(x)
    end
end

@show typeof(g3(x))
@show typeof(g3(x2));  # type stable
```

```text
typeof(g2(x)) = BigFloat
typeof(g2(x2)) = Nothing
typeof(g3(x)) = BigFloat
typeof(g3(x2)) = Nothing
```

These patterns are relatively straightforward, but generic programming can be thought of as a Leontief production function: if any of the functions you write or call are not precise enough, then it may break the chain.

This is all the more reason to exploit carefully designed packages rather than “do-it-yourself”.

6.6.6 A Digression on Style and Naming

The previous section helps to establish some of the reasoning behind the style choices in these lectures: “be aware of types, but avoid declaring them”.

The purpose of this is threefold:

- Provide easy to read code with minimal “syntactic noise” and a clear correspondence to the math.
- Ensure that code is sufficiently generic to exploit other packages and types.
- Avoid common mistakes and unnecessary performance degradations.

This is just one of many decisions and patterns to ensure that your code is consistent and clear.

The best resource is to carefully read other peoples code, but a few sources to review are

- Julia Style Guide.
- Invenia Blue Style Guide.
- Julia Praxis Naming Guides.
- QuantEcon Style Guide used in these lectures.

Now why would we emphasize naming and style as a crucial part of the lectures?
Because it is an essential tool for creating research that is **reproducible** and **correct**.

Some helpful ways to think about this are

- **Clearly written code is easier to review for errors**: The first-order concern of any code is that it correctly implements the whiteboard math.
- **Code is read many more times than it is written**: Saving a few keystrokes in typing a variable name is never worth it, nor is a divergence from the mathematical notation where a single symbol for a variable name would map better to the model.
- **Write code to be read in the future, not today**: If you are not sure anyone else will read the code, then write it for an ignorant future version of yourself who may have forgotten everything, and is likely to misuse the code.
- **Maintain the correspondence between the whiteboard math and the code**: For example, if you change notation in your model, then immediately update all variables in the code to reflect it.

### Commenting Code

One common mistake people make when trying to apply these goals is to add in a large number of comments.

Over the years, developers have found that excess comments in code (and especially big comment headers used before every function declaration) can make code harder to read.

The issue is one of syntactic noise: if most of the comments are redundant given clear variable and function names, then the comments make it more difficult to mentally parse and read the code.

If you examine Julia code in packages and the core language, you will see a great amount of care taken in function and variable names, and comments are only added where helpful.

For creating packages that you intend others to use, instead of a comment header, you should use **docstrings**.

### 6.7 Introduction to Multiple Dispatch

One of the defining features of Julia is **multiple dispatch**, whereby the same function name can do different things depending on the underlying types.

Without realizing it, in nearly every function call within packages or the standard library you have used this feature.

To see this in action, consider the absolute value function **abs**

```julia
In [39]: @show abs(-1)  # `Int64`
    @show abs(-1.0)  # `Float64`
    @show abs(0.0 - 1.0im);  # `Complex{Float64}`

    abs(-1) = 1
    abs(-1.0) = 1.0
    abs(0.0 - 1.0im) = 1.0
```

In all of these cases, the **abs** function has specialized code depending on the type passed in.
To do this, a function specifies different **methods** which operate on a particular set of types. Unlike most cases we have seen before, this requires a type annotation.

To rewrite the `abs` function

In [40]:

```python
function ourabs(x::Real)
    if x > zero(x)  # note, not 0!
        return x
    else
        return -x
    end
end

function ourabs(x::Complex)
    sqrt(real(x)^2 + imag(x)^2)
end

@show ourabs(-1)  # `Int64`
@show ourabs(-1.0) # `Float64`
@show ourabs(1.0 - 2.0im);  # `Complex{Float64}`

ourabs(-1) = 1
ourabs(-1.0) = 1.0
ourabs(1.0 - 2.0im) = 2.23606797749979
```

Note that in the above, `x` works for any type of `Real`, including `Int64`, `Float64`, and ones you may not have realized exist

In [41]:

```python
x = -2//3  # `Rational` number, -2/3
@show typeof(x)
@show ourabs(x);

    typeof(x) = Rational{Int64}
ourabs(x) = 2//3
```

You will also note that we used an abstract type, `Real`, and an incomplete parametric type, `Complex`, when defining the above functions.

Unlike the creation of `struct` fields, there is no penalty in using abstract types when you define function parameters, as they are used purely to determine which version of a function to use.

### 6.7.1 Multiple Dispatch in Algorithms (Advanced)

If you want an algorithm to have specialized versions when given different input types, you need to declare the types for the function inputs.

As an example where this could come up, assume that we have some grid `x` of values, the results of a function `f` applied at those values, and want to calculate an approximate derivative using forward differences.

In that case, given $x_n, x_{n+1}, f(x_n)$ and $f(x_{n+1})$, the forward-difference approximation of the derivative is
\[ f'(x_n) \approx \frac{f(x_{n+1}) - f(x_n)}{x_{n+1} - x_n} \]

To implement this calculation for a vector of inputs, we notice that there is a specialized implementation if the grid is uniform.

The uniform grid can be implemented using an `AbstractRange`, which we can analyze with `typeof`, `supertype` and `show_supertypes`.

In [42]:
```plaintext
x = range(0.0, 1.0, length = 20)
x_2 = 1:1:20  # if integers
@show typeof(x)
@show typeof(x_2)
@show supertype(typeof(x))
```

```
typeof(x) = StepRangeLen{Float64,Base.TwicePrecision{Float64},Base.TwicePrecision{Float64}}
typeof(x_2) = StepRange{Int64,Int64}
supertype(typeof(x)) = AbstractRange{Float64}
```

Out[42]: `AbstractRange{Float64}`

To see the entire tree about a particular type, use `show_supertypes`.

In [43]: `show_supertypes(typeof(x))`  # or typeof(x) |> show_supertypes

```
StepRangeLen{Float64,Base.TwicePrecision{Float64},Base.TwicePrecision{Float64}} <: AbstractRange{Float64} <: AbstractArray{Float64,1} <: Any
```

In [44]: `show_supertypes(typeof(x_2))`

```
StepRange{Int64,Int64} <: OrdinalRange{Int64,Int64} <: AbstractRange{Int64} <: AbstractArray{Int64,1} <: Any
```

The types of the range objects can be very complicated, but are both subtypes of `AbstractRange`.

In [45]: `@show typeof(x) <: AbstractRange`
   `@show typeof(x_2) <: AbstractRange;`

```
typeof(x) <: AbstractRange = true
typeof(x_2) <: AbstractRange = true
```

While you may not know the exact concrete type, any `AbstractRange` has an informal set of operations that are available.
Similarly, there are a number of operations available for any AbstractVector, such as length.

In [47]: f(x) = x^2
   f_x = f.(x)  # calculating at the range values
   typeof(f_x)
supertypes(typeof(f_x))
supertypes(supertypes(typeof(f_x)))  # walk up tree again!
   length(f_x);  # and many more

   typeof(f_x) = Array{Float64,1}
supertypes(typeof(f_x)) = DenseArray{Float64,1}
supertypes(supertypes(typeof(f_x))) = AbstractArray{Float64,1}
   length(f_x) = 20

In [48]: show_supertypes(typeof(f_x))

   Array{Float64,1} <: DenseArray{Float64,1} <: AbstractArray{Float64,1} <: Any

There are also many functions that can use any AbstractArray, such as diff.

?diff
search: diff symdiff setdiff symdiff! setdiff! Cptrdiff_t
diff(A::AbstractVector)  # finite difference operator of matrix or vector A
# if A is a matrix, specify the dimension over which to operate with the dims keyword argument
diff(A::AbstractMatrix; dims::Integer)

Hence, we can call this function for anything of type AbstractVector.

Finally, we can make a high performance specialization for any AbstractVector and AbstractRange.

In [49]: slopes(f_x::AbstractVector, x::AbstractRange) = diff(f_x) / step(x)

Out[49]: slopes (generic function with 1 method)
We can use auto-differentiation to compare the results.

```
In [50]: using Plots, ForwardDiff
   gr(fmt = :png);

   # operator to get the derivative of this function using AD
   D(f) = x -> ForwardDiff.derivative(f, x)

   # compare slopes with AD for sin(x)
   q(x) = sin(x)
   x = 0.0:0.1:4.0
   q_x = q.(x)
   q_slopes_x = slopes(q_x, x)

   D_q_x = D(q).(x)  # broadcasts AD across vector

   plot(x[1:end-1], D_q_x[1:end-1], label = "q' with AD")
   plot!(x[1:end-1], q_slopes_x, label = "q slopes")
```

```
Out[50]:
```

Consider a variation where we pass a function instead of an AbstractArray

```
In [51]: slopes(f::Function, x::AbstractRange) = diff(f.(x)) / step(x)  # broadcast function

@show typeof(q) <: Function
@show typeof(x) <: AbstractRange
q_slopes_x = slopes(q, x)  # use slopes(f::Function, x)
@show q_slopes_x[1];
```
Finally, if \( x \) was an `AbstractArray` and not an `AbstractRange` we can no longer use a uniform step.

For this, we add in a version calculating slopes with forward first-differences

```python
In [52]: # broadcasts over the diff
   ...: slopes(f::Function, x::AbstractArray) = diff(f.(x)) ./ diff(x)
   ...: x_array = Array(x)  # convert range to array
   ...: @show typeof(x_array) <: AbstractArray
   ...: q_slopes_x = slopes(q, x_array)
   ...: @show q_slopes_x[1];

   ...:
   ...: typeof(x_array) <: AbstractArray = true
   ...: q_slopes_x[1] = 0.9983341664682815
```

In the final example, we see that it is able to use specialized implementations over both the \( f \) and the \( x \) arguments.

This is the “multiple” in multiple dispatch.

### 6.8 Exercises

#### 6.8.1 Exercise 1

Explore the package `StaticArrays.jl`.

- Describe two abstract types and the hierarchy of three different concrete types.
- Benchmark the calculation of some simple linear algebra with a static array compared to the following for a dense array for \( N = 3 \) and \( N = 15 \).

```python
In [53]: using BenchmarkTools
   ...:
   ...: N = 3
   ...: A = rand(N, N)
   ...: x = rand(N)
   ...: @btime $A * $x  # the $ in front of variable names is sometimes important
   ...: @btime inv($A)

   ...: 84.900 ns (1 allocation: 112 bytes)
   ...: 778.746 ns (5 allocations: 1.98 KiB)
```

```plaintext
Out[53]:
3×3 Array{Float64,2}:
0.69559 2.62482 -5.72178
1.46424 -4.30108 7.48458
-1.66542 3.15164 -2.70676
```
6.8.2 Exercise 2

A key step in the calculation of the Kalman Filter is calculation of the Kalman gain, as can be seen with the following example using dense matrices from the Kalman lecture.

Using what you learned from Exercise 1, benchmark this using Static Arrays

\[
\Sigma = \begin{bmatrix}
0.4 & 0.3 \\
0.3 & 0.45
\end{bmatrix}
\]

\[G = I\]

\[R = 0.5 \times \Sigma\]

\[\text{gain}(\Sigma, G, R) = \Sigma \times G' \times \text{inv}(G \times \Sigma \times G' + R)\]

\[
\text{@btime} \quad \text{gain}(\$\Sigma, \$G, \$R)
\]

950.875 ns (10 allocations: 1.94 KiB)

Out[54]: 2×2 Array{Float64,2}:
0.666667 1.11022e-16
1.11022e-16 0.666667

How many times faster are static arrays in this example?

6.8.3 Exercise 3

The Polynomial.jl provides a package for simple univariate Polynomials.

In [55]: using Polynomials

\[p = \text{Polynomial}(\begin{bmatrix}2 & -5 & 2\end{bmatrix}, :x) \quad \# :x \text{ just gives a symbol for display}\]

\[\text{@show} \quad p\]

\[p' = \text{derivative}(p) \quad \# \text{ gives the derivative of } p, \text{ another polynomial}\]

\[\text{@show} \quad p(0.1), p'(0.1) \quad \# \text{ call like a function}\]

\[\text{@show} \quad \text{roots}(p); \quad \# \text{ find roots such that } p(x) = 0\]

\[p = \text{Polynomial}(2 - 5x + 2x^2)\]

\[(p(0.1), p'(0.1)) = (1.52, -4.6)\]

\[\text{roots}(p) = [0.5, 2.0]\]

Plot both \(p(x)\) and \(p'(x)\) for \(x \in [-2,2]\).

6.8.4 Exercise 4

Use your solution to Exercise 8(a/b) in Introductory Examples to create a specialized version of Newton’s method for Polynomials using the derivative function.

The signature of the function should be \text{newtonsmethod}(p::Polynomial, x_0; tolerance = 1E-7, maxiter = 100), where \(p::\text{Polynomial}\) ensures that this version of the function will be used anytime a polynomial is passed (i.e. dispatch).

Compare the results of this function to the built-in \text{roots}(p) function.
6.8.5 Exercise 5 (Advanced)

The trapezoidal rule approximates an integral with

\[
\int_{\underline{x}}^{\overline{x}} f(x) \, dx \approx \sum_{n=1}^{N} \frac{f(x_{n-1}) + f(x_n)}{2} \Delta x_n
\]

where \(x_0 = \underline{x}, \ x_N = \overline{x}\), and \(\Delta x_n \equiv x_{n-1} - x_n\).

Given an \(x\) and a function \(f\), implement a few variations of the trapezoidal rule using multiple dispatch

- \(\text{trapezoidal}(f, x)\) for any \(\text{typeof}(x) = \text{AbstractArray}\) and \(\text{typeof}(f) == \text{AbstractArray}\) where \(\text{length}(x) = \text{length}(f)\)
- \(\text{trapezoidal}(f, x)\) for any \(\text{typeof}(x) = \text{AbstractRange}\) and \(\text{typeof}(f) == \text{AbstractArray}\) where \(\text{length}(x) = \text{length}(f)\)
  - Exploit the fact that \(\text{AbstractRange}\) has constant step sizes to specialize the algorithm
- \(\text{trapezoidal}(f, \underline{x}, \overline{x}, N)\) where \(\text{typeof}(f) = \text{Function}\), and the other arguments are \(\text{Real}\)
  - For this, build a uniform grid with \(N\) points on \([\underline{x}, \overline{x}]\) – call the \(f\) function at those grid points and use the existing \(\text{trapezoidal}(f, x)\) from the implementation

With these: 1. Test each variation of the function with \(f(x) = x^2\) with \(\underline{x} = 0, \overline{x} = 1\). 2. From the analytical solution of the function, plot the error of \(\text{trapezoidal}(f, \underline{x}, \overline{x}, N)\) relative to the analytical solution for a grid of different \(N\) values. 3. Consider trying different functions for \(f(x)\) and compare the solutions for various \(N\).

When trying different functions, instead of integrating by hand consider using a high-accuracy library for numerical integration such as QuadGK.jl

In [56]: using QuadGK

\[
f(x) = x^2
\]

\[
\text{value, accuracy} = \text{quadgk}(f, 0.0, 1.0)
\]

Out[56]: \((0.3333333333333333, 5.551115123125783\times10^{-17})\)

6.8.6 Exercise 6 (Advanced)

Take a variation of your code in Exercise 5.

Use auto-differentiation to calculate the following derivative for the example functions

\[
\frac{d}{dx} \int_{\underline{x}}^{\overline{x}} f(x) \, dx
\]

Hint: See the following code for the general pattern, and be careful to follow the rules for generic programming.
In [57]: using ForwardDiff

    function f(a, b; N = 50)
        r = range(a, b, length=N) # one
        return mean(r)
    end

    Df(x) = ForwardDiff.derivative(y -> f(0.0, y), x)

@show f(0.0, 3.0)
@show f(0.0, 3.1)

    Df(3.0)

    f(0.0, 3.0) = 1.5
    f(0.0, 3.1) = 1.55

Out[57]: 0.5
Part II

Packages and Software Engineering in Julia
Chapter 7

Generic Programming

7.1 Contents

- Overview 7.2
- Exploring Type Trees 7.3
- Distributions 7.4
- Numbers and Algebraic Structures 7.5
- Reals and Algebraic Structures 7.6
- Functions, and Function-Like Types 7.7
- Limitations of Dispatching on Abstract Types 7.8
- Exercises 7.9

I find OOP methodologically wrong. It starts with classes. It is as if mathematicians would start with axioms. You do not start with axioms - you start with proofs. Only when you have found a bunch of related proofs, can you come up with axioms. You end with axioms. The same thing is true in programming: you have to start with interesting algorithms. Only when you understand them well, can you come up with an interface that will let them work. – Alexander Stepanov

7.2 Overview

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

- abstract and concrete types
- the type tree
- designing and using generic interfaces
- the role of generic interfaces in Julia performance

Understanding them will help you

- form a “mental model” of the Julia language
- design code that matches the “white-board” mathematics
- create code that can use (and be used by) a variety of other packages
- write “well organized” Julia code that’s easy to read, modify, maintain and debug
- improve the speed at which your code runs

(Special thank you to Jeffrey Sarnoff)
7.2.1 Generic Programming is an Attitude

From *Mathematics to Generic Programming* [99]

Generic programming is an approach to programming that focuses on designing algorithms and data structures so that they work in the most general setting without loss of efficiency... Generic programming is more of an attitude toward programming than a particular set of tools.

In that sense, it is important to think of generic programming as an interactive approach to uncover generality without compromising performance rather than as a set of rules.

As we will see, the core approach is to treat data structures and algorithms as loosely coupled, and is in direct contrast to the is-a approach of object-oriented programming.

This lecture has the dual role of giving an introduction into the design of generic algorithms and describing how Julia helps make that possible.

7.2.2 Setup

In [1]: using InstantiateFromURL
   # optionally add arguments to force installation: instantiate = true,
   precompile = true
   github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

In [2]: using LinearAlgebra, Statistics
   using Distributions, Plots, QuadGK, Polynomials, Interpolations
   gr(fmt = :png);

7.3 Exploring Type Trees

The connection between data structures and the algorithms which operate on them is handled by the type system.

Concrete types (i.e., `Float64` or `Array{Float64, 2}`) are the data structures we apply an algorithm to, and the abstract types (e.g. the corresponding `Number` and `AbstractArray`) provide the mapping between a set of related data structures and algorithms.

In [3]: using Distributions
   x = 1
   y = Normal()
   z = "foo"
   @show x, y, z
   @show typeof(x), typeof(y), typeof(z)
   @show supertype(typeof(x))
   # pipe operator, |>, is is equivalent
   @show typeof(x) |> supertype
   @show supertype(typeof(y))
   @show typeof(z) |> supertype
   @show typeof(x) <: Any;
7.3. EXPLORING TYPE TREES

```
(x, y, z) = (1, Normal{Float64}(μ=0.0, σ=1.0), "foo")
(typeof(x), typeof(y), typeof(z)) = (Int64, Normal{Float64}, String)
supertype(typeof(x)) = Signed
    typeof(x) |> supertype = Signed
supertype(typeof(y)) = Distribution{Univariate,Continuous}
typeof(z) |> supertype = AbstractString
typeof(x) <: Any = true
```

Beyond the `typeof` and `supertype` functions, a few other useful tools for analyzing the tree of types are discussed in the introduction to types lecture.

In [4]: using Base: show_supertypes # import the function from the `Base` package

```
show_supertypes(Int64)

    Int64 <: Signed <: Integer <: Real <: Number <: Any
```

In [5]: subtypes(Integer)

```
Out[5]: 4-element Array{Any,1}:
    Bool
    GeometryTypes.OffsetInteger
    Signed
    Unsigned
```

Using the `subtypes` function, we can write an algorithm to traverse the type tree below any time `t` – with the confidence that all types support `subtypes`.

In [6]: # from https://github.com/JuliaLang/julia/issues/24741

```
function subtypetree(t, level=1, indent=4)
    if level == 1
        println(t)
    end
    for s in subtypes(t)
        println(join(fill(" ", level * indent)) * string(s)) # print type
        subtypetree(s, level+1, indent) # recursively print the next type,
        indenting
    end
end
```

```
Out[6]: subtypetree (generic function with 3 methods)
```

Applying this to `Number`, we see the tree of types currently loaded.

In [7]: subtypetree(Number) # warning: do not use this function on `\`Any``

```
Number
    Complex
    Real
        AbstractFloat
        BigFloat
```
Float16
Float32
Float64
AbstractIrrational
Irrational
FixedPointNumbers.FixedPoint
FixedPointNumbers.Fixed
FixedPointNumbers.Normed
Integer
Bool
GeometryTypes.OffsetInteger
Signed
BigInt
Int128
Int16
Int32
Int64
Int8
Unsigned
UInt128
UInt16
UInt32
UInt64
UInt8
Rational
Ratios.SimpleRatio
StatsBase.TestStat

For the most part, all of the “leaves” will be concrete types.

### 7.3.1 Any

At the root of all types is Any

There are a few functions that work in the “most generalized” context: usable with anything that you can construct or access from other packages.

We have already called `typeof`, `show` and `supertype` – which will apply to a custom `struct` type since `MyType <: Any`

```plaintext
In [8]: # custom type
    struct MyType
        a::Float64
    end

myval = MyType(2.0)
@show myval
@show typeof(myval)
@show supertype(typeof(myval))
@show typeof(myval) <: Any;

myval = MyType(2.0)
typeof(myval) = MyType
supertype(typeof(myval)) = Any
typeof(myval) <: Any = true
```
7.3. EXPLORING TYPE TREES

Here we see another example of generic programming: every type `<: Any supports the @show macro, which in turn, relies on the show function.

The @show macro (1) prints the expression as a string; (2) evaluates the expression; and (3) calls the show function on the returned values.

To see this with built-in types

In [9]: x = [1, 2]
    show(x)

    [1, 2]

The Any type is useful, because it provides a fall-back implementation for a variety of functions.

Hence, calling show on our custom type dispatches to the fallback function

In [10]: myval = MyType(2.0)
    show(myval)

    MyType(2.0)

The default fallback implementation used by Julia would be roughly equivalent to

```julia
function show(io::IO, x)
    str = string(x)
    print(io, str)
end
```

To implement a specialized implementation of the show function for our type, rather than using this fallback

In [11]: import Base.show # to extend an existing function

    function show(io::IO, x::MyType)
        str = "(MyType.a = $(x.a))" # custom display
        print(io, str)
    end
    show(myval) # it creates an IO value first and then calls the above show

    (MyType.a = 2.0)

At that point, we can use the @show macro, which in turn calls show

In [12]: @show myval;

    myval = (MyType.a = 2.0)

Here we see another example of generic programming: any type with a show function works with @show.

Layering of functions (e.g. @show calling show) with a “fallback” implementation makes it possible for new types to be designed and only specialized where necessary.
7.3.2 Unlearning Object Oriented (OO) Programming (Advanced)

See Types for more on OO vs. generic types.

If you have never used programming languages such as C++, Java, and Python, then the type hierarchies above may seem unfamiliar and abstract.

In that case, keep an open mind that this discussion of abstract concepts will have practical consequences, but there is no need to read this section.

Otherwise, if you have used object-oriented programming (OOP) in those languages, then some of the concepts in these lecture notes will appear familiar.

Don’t be fooled!

The superficial similarity can lead to misuse: types are not classes with poor encapsulation, and methods are not the equivalent to member functions with the order of arguments swapped.

In particular, previous OO knowledge often leads people to write Julia code such as

```julia
# BAD! Replicating an OO design in Julia
mutable struct MyModel
    a::Float64
    b::Float64
    algorithmcalculation::Float64

    MyModel(a, b) = new(a, b, 0.0) # an inner constructor
end

function myalgorithm!(m::MyModel, x)
    m.algorithmcalculation = m.a + m.b + x # some algorithm
end

function set_a!(m::MyModel, a)
    m.a = a
end

m = MyModel(2.0, 3.0)
x = 0.1
set_a!(m, 4.1)
myalgorithm!(m, x)
@show m.algorithmcalculation;

m.algorithmcalculation = 7.199999999999999
```

You may think to yourself that the above code is similar to OO, except that you * reverse the first argument, i.e., `myalgorithm!(m, x)` instead of the object-oriented `m.myalgorithm!(x)` * cannot control encapsulation of the fields `a`, `b`, but you can add getter/setters like `set_a` * do not have concrete inheritance

While this sort of programming is possible, it is (verbosely) missing the point of Julia and the power of generic programming.

When programming in Julia

- there is no encapsulation and most custom types you create will be immutable.
7.4. DISTRIBUTIONS

- **Polymorphism** is achieved without anything resembling OOP inheritance.
- **Abstraction** is implemented by keeping the data and algorithms that operate on them as orthogonal as possible – in direct contrast to OOP’s association of algorithms and methods directly with a type in a tree.
- The supertypes in Julia are simply used for selecting which specialized algorithm to use (i.e., part of generic polymorphism) and have nothing to do with OO inheritance.
- The looseness that accompanies keeping algorithms and data structures as orthogonal as possible makes it easier to discover commonality in the design.

**Iterative Design of Abstractions**

As its essence, the design of generic software is that you will start with creating algorithms which are largely orthogonal to concrete types.

In the process, you will discover commonality which leads to abstract types with informally defined functions operating on them.

Given the abstract types and commonality, you then refine the algorithms as they are more limited or more general than you initially thought.

This approach is in direct contrast to object-oriented design and analysis (OOAD).

With that, where you specify a taxonomies of types, add operations to those types, and then move down to various levels of specialization (where algorithms are embedded at points within the taxonomy, and potentially specialized with inheritance).

In the examples that follow, we will show for exposition the hierarchy of types and the algorithms operating on them, but the reality is that the algorithms are often designed first, and the abstract types came later.

**7.4 Distributions**

First, consider working with “distributions”.

Algorithms using distributions might (1) draw random numbers for Monte-Carlo methods; and (2) calculate the pdf or cdf – if it is defined.

The process of using concrete distributions in these sorts of applications led to the creation of the `Distributions.jl` package.

Let’s examine the tree of types for a Normal distribution

```plaintext
In [14]: using Distributions
    d1 = Normal(1.0, 2.0) # an example type to explore
    @show d1
    show_supertypes(typeof(d1))

    d1 = Normal{Float64}({μ=1.0, σ=2.0})
    Normal{Float64} <: Distribution{Univariate,Continuous} <: Sampleable{Univariate,Continuous} <: Any
```

The `Sampleable{Univariate,Continuous}` type has a limited number of functions, chiefly the ability to draw a random number
In [15]: @show rand(d1);

    rand(d1) = -3.8524551916467678

The purpose of that abstract type is to provide an interface for drawing from a variety of distributions, some of which may not have a well-defined predefined pdf.

If you were writing a function to simulate a stochastic process with arbitrary iid shocks, where you did not need to assume an existing pdf etc., this is a natural candidate.

For example, to simulate $x_{t+1} = ax_t + b\epsilon_{t+1}$ where $\epsilon \sim D$ for some $D$, which allows drawing random values.

In [16]: function simulateprocess(x; a = 1.0, b = 1.0, N = 5, 
d::Sampleable{Univariate,Continuous})
    x = zeros(typeof(x), N+1) # preallocate vector, careful on the type
    x[1] = x
    for t in 2:N+1
        x[t] = a * x[t-1] + b * rand(d) # draw
    end
    return x
end
@show simulateprocess(0.0, d = Normal(0.2, 2.0));

simulateprocess(0.0, d = Normal(0.2, 2.0)) = [0.0, -2.2517753336924105, 
-1.499729413286212, -3.5981086351011697, -2.892126808220728, -5.758551572845958]

The `Sampleable{Univariate,Continuous}` and, especially, the `Sampleable{Multivariate,Continuous}` abstract types are useful generic interfaces for Monte-Carlo and Bayesian methods.

Moving down the tree, the `Distributions{Univariate, Continuous}` abstract type has other functions we can use for generic algorithms operating on distributions.

These match the mathematics, such as `pdf`, `cdf`, `quantile`, `support`, `minimum`, `maximum`, etc.

In [17]: d1 = Normal(1.0, 2.0) 
d2 = Exponential(0.1) 
@show d1 
@show d2 
@show supertype(typeof(d1)) 
@show supertype(typeof(d2)) 

@show pdf(d1, 0.1) 
@show pdf(d2, 0.1) 
@show cdf(d1, 0.1) 
@show cdf(d2, 0.1) 
@show support(d1) 
@show support(d2) 
@show minimum(d1) 
@show minimum(d2) 
@show maximum(d1) 
@show maximum(d2);
d1 = Normal(Float64)(μ=1.0, σ=2.0)
d2 = Exponential(Float64)(θ=0.1)
supertype(typeof(d1)) = Distribution{Univariate,Continuous}
supertype(typeof(d2)) = Distribution{Univariate,Continuous}

pdf(d1, 0.1) = 0.18026348123082397
pdf(d2, 0.1) = 3.6787944117144233
cdf(d1, 0.1) = 0.32635522028792
cdf(d2, 0.1) = 0.6321205588285577

You could create your own `Distributions{Univariate, Continuous}` type by implementing those functions – as is described in the documentation.

If you fulfill all of the conditions of a particular interface, you can use algorithms from the present, past, and future that are written for the abstract `Distributions{Univariate, Continuous}` type.

As an example, consider the StatsPlots package

In [18]: using StatsPlots
   d = Normal(2.0, 1.0)
   plot(d) # note no other arguments!

Out[18]:

Calling `plot` on any subtype of `Distributions{Univariate, Continuous}` displays the `pdf` and uses `minimum` and `maximum` to determine the range.

Let’s create our own distribution type
In [19]: struct OurTruncatedExponential <: Distribution{Univariate,Continuous}
    α::Float64
    xmax::Float64
end

Distributions.pdf(d::OurTruncatedExponential, x) = d.α * \exp(-d.α * x)/
\exp(-d.α * d.xmax)

Distributions.minimum(d::OurTruncatedExponential) = 0
Distributions.maximum(d::OurTruncatedExponential) = d.xmax
# ... more to have a complete type

To demonstrate this

In [20]: d = OurTruncatedExponential(1.0, 2.0)
@show minimum(d), maximum(d)
@show support(d) # why does this work?

    (minimum(d), maximum(d)) = (0.0, 2.0)
support(d) = RealInterval(0.0, 2.0)

Out[20]: RealInterval(0.0, 2.0)

Curiously, you will note that the support function works, even though we did not provide one.

This is another example of the power of multiple dispatch and generic programming.

In the background, the Distributions.jl package has something like the following implemented

    Distributions.support(d::Distribution) = RealInterval(minimum(d), maximum(d))

Since OurTruncatedExponential <: Distribution, and we implemented minimum and maximum, calls to support get this implementation as a fallback.

These functions are enough to use the StatsPlots.jl package

In [21]: plot(d) # uses the generic code!

Out[21]:
A few things to point out

- Even if it worked for StatsPlots, our implementation is incomplete, as we haven’t fulfilled all of the requirements of a Distribution.
- We also did not implement the rand function, which means we are breaking the implicit contract of the Sampleable abstract type.
- It turns out that there is a better way to do this precise thing already built into Distributions.

```python
In [22]: d = Truncated(Exponential(0.1), 0.0, 2.0)
@show typeof(d)
plot(d)

typeof(d) = Truncated{Exponential{Float64},Continuous,Float64}
```

Out[22]:
This is the power of generic programming in general, and Julia in particular: you can combine and compose completely separate packages and code, as long as there is an agreement on abstract types and functions.

### 7.5 Numbers and Algebraic Structures

Define two binary functions, $+$ and $\cdot$, called addition and multiplication – although the operators can be applied to data structures much more abstract than a Real.

In mathematics, a ring is a set with associated additive and multiplicative operators where

- the additive operator is associative and commutative
- the multiplicative operator is associative and distributive with respect to the additive operator
- there is an additive identity element, denoted $0$, such that $a + 0 = a$ for any $a$ in the set
- there is an additive inverse of each element, denoted $-a$, such that $a + (-a) = 0$
- there is a multiplicative identity element, denoted $1$, such that $a \cdot 1 = a = 1 \cdot a$
- a total or partial ordering is not required (i.e., there does not need to be any meaningful $<$ operator defined)
- a multiplicative inverse is not required

While this skips over some parts of the mathematical definition, this algebraic structure provides motivation for the abstract Number type in Julia

- **Remark:** We use the term “motivation” because they are not formally connected and the mapping is imperfect.

- The main difficulty when dealing with numbers that can be concretely created on a computer is that the requirement that the operators are closed in the set are difficult to ensure (e.g. floating points have finite numbers of bits of information).
Let \( \text{typeof}(a) = \text{typeof}(b) = T <: \text{Number} \), then under an informal definition of the \textbf{generic interface} for \text{Number}, the following must be defined

- the additive operator: \( a + b \)
- the multiplicative operator: \( a * b \)
- an additive inverse operator: \( -a \)
- an inverse operation for addition \( a - b = a + (-b) \)
- an additive identity: \( \text{zero}(T) \) or \( \text{zero}(a) \) for convenience
- a multiplicative identity: \( \text{one}(T) \) or \( \text{one}(a) \) for convenience

The core of generic programming is that, given the knowledge that a value is of type \text{Number}, we can design algorithms using any of these functions and not concern ourselves with the particular concrete type.

Furthermore, that generality in designing algorithms comes with no compromises on performance compared to carefully designed algorithms written for that particular type.

To demonstrate this for a complex number, where \text{Complex\{Float64\} <: Number}:

\[
\begin{align*}
\text{In [23]:} & \quad a = 1.0 + 1.0\text{im} \\
& \quad b = 0.0 + 2.0\text{im} \\
& \quad \text{typeof}(a) = \text{Complex\{Float64\}} \\
& \quad \text{typeof}(a) <: \text{Number} = \text{true} \\
& \quad a + b = 1.0 + 3.0\text{im} \\
& \quad a * b = -2.0 + 2.0\text{im} \\
& \quad -a = -1.0 - 1.0\text{im} \\
& \quad a - b = 1.0 - 1.0\text{im} \\
& \quad \text{zero}(a) = 0.0 + 0.0\text{im} \\
& \quad \text{one}(a) = 1.0 + 0.0\text{im}
\end{align*}
\]

And for an arbitrary precision integer where \text{BigInt <: Number} (i.e., a different type than the \text{Int64} you have worked with, but nevertheless a \text{Number})

\[
\begin{align*}
\text{In [24]:} & \quad a = \text{BigInt}(10) \\
& \quad b = \text{BigInt}(4) \\
& \quad \text{typeof}(a) = \text{BigInt} \\
& \quad \text{typeof}(a) <: \text{Number} = \text{true} \\
& \quad a + b = 14 \\
& \quad a * b = 40 \\
& \quad -a = -10 \\
& \quad a - b = 6 \\
& \quad \text{zero}(a) = 0 \\
& \quad \text{one}(a) = 1
\end{align*}
\]
7.5.1 Complex Numbers and Composition of Generic Functions

This allows us to showcase further how different generic packages compose – even if they are only loosely coupled through agreement on common generic interfaces.

The Complex numbers require some sort of storage for their underlying real and imaginary parts, which is itself left generic.

This data structure is defined to work with any type <: Number, and is parameterized (e.g. Complex{Float64} is a complex number storing the imaginary and real parts in Float64).

In [25]: x = 4.0 + 1.0im
  @show x, typeof(x)
   xbig = BigFloat(4.0) + 1.0im
  @show xbig, typeof(xbig);

   (x, typeof(x)) = (4.0 + 1.0im, Complex{Float64})
   (xbig, typeof(xbig)) = (4.0 + 1.0im, Complex{BigFloat})

The implementation of the Complex numbers use the underlying operations of storage type, so as long as +, * etc. are defined – as they should be for any Number – the complex operation can be defined

In [26]: @which +(x, x)

Out[26]: +(z::Complex, w::Complex) in Base at complex.jl:275

Following that link, the implementation of + for complex numbers is

+(z::Complex, w::Complex) = Complex(real(z) + real(w), imag(z) + imag(w))

real(z) and imag(z) returns the associated components of the complex number in the underlying storage type (e.g. Float64 or BigFloat).

The rest of the function has been carefully written to use functions defined for any Number (e.g. + but not <, since it is not part of the generic number interface).

To follow another example, look at the implementation of abs specialized for complex numbers

In [27]: @which abs(x)

Out[27]: abs(z::Complex) in Base at complex.jl:264
The source is

\[
\text{abs}(z :: \text{Complex}) = \text{hypot}(\text{real}(z), \text{imag}(z))
\]

In this case, if you look at the generic function to get the hypotenuse, \text{hypot}, you will see that it has the function signature \text{hypot}(x :: T, y :: T) where T <: \text{Number}, and hence works for any \text{Number}.

That function, in turn, relies on the underlying \text{abs} for the type of \text{real}(z).

This would dispatch to the appropriate \text{abs} for the type

\[
\text{In [28]: @which abs(1.0)}
\]

\[
\text{Out[28]: abs(x :: Float64) in Base at float.jl:528}
\]

\[
\text{In [29]: @which abs(BigFloat(1.0))}
\]

\[
\text{Out[29]: abs(x :: Real) in Base at number.jl:120}
\]

With implementations

\[
\text{abs(x :: Real) = ifelse(signbit(x), -x, x)}
\]

\[
\text{abs(x :: Float64) = abs(float(x))}
\]

For a \text{Real} number (which we will discuss in the next section) the fallback implementation calls a function \text{signbit} to determine if it should flip the sign of the number.

The specialized version for \text{Float64 <: Real} calls a function called \text{abs_float} – which turns out to be a specialized implementation at the compiler level.

While we have not completely dissected the tree of function calls, at the bottom of the tree you will end at the most optimized version of the function for the underlying datatype.

Hopefully this showcases the power of generic programming: with a well-designed set of abstract types and functions, the code can both be highly general and composable and still use the most efficient implementation possible.

### 7.6 Reals and Algebraic Structures

Thinking back to the mathematical motivation, a field is a ring with a few additional properties, among them

- a multiplicative inverse: \( a^{-1} \)
- an inverse operation for multiplication: \( a/b = a \cdot b^{-1} \)

Furthermore, we will make it a total ordered field with

- a total ordering binary operator: \( a < b \)
This type gives some motivation for the operations and properties of the Real type.

Of course, Complex{Float64} <: Number but not Real – since the ordering is not defined for complex numbers in mathematics.

These operations are implemented in any subtype of Real through

- the multiplicative inverse: inv(a)
- the multiplicative inverse operation: a / b = a * inv(b)
- an ordering a < b

We have already shown these with the Float64 and BigFloat.

To show this for the Rational number type, where a // b constructs a rational number \( \frac{a}{b} \)

In [30]:
```julia
a = 1 // 10
b = 4 // 6
@show typeof(a)
@show typeof(a) <: Number
@show typeof(a) <: Real
@show inv(a)
@show a / b
@show a < b;
```

```
typeof(a) = Rational{Int64}
typeof(a) <: Number = true
typeof(a) <: Real = true
inv(a) = 10//1
a / b = 3//20
a < b = true
```

**Remark:** Here we see where and how the precise connection to the mathematics for number types breaks down for practical reasons, in particular

- Integer types (i.e., Int64 <: Integer) do not have a a multiplicative inverse with closure in the set.

- However, it is necessary in practice for integer division to be defined, and return back a member of the Real's.
- This is called type promotion, where a type can be converted to another to ensure an operation is possible by direct conversion between types (i.e., it can be independent of the type hierarchy).

Do not think of the break in the connection between the underlying algebraic structures and the code as a failure of the language or design.

Rather, the underlying algorithms for use on a computer do not perfectly fit the algebraic structures in this instance.

Moving further down the tree of types provides more operations more directly tied to the computational implementation than abstract algebra.

For example, floating point numbers have a machine precision, below which numbers become indistinguishable due to lack of sufficient “bits” of information.
In [31]: @show Float64 <: AbstractFloat
@show BigFloat <: AbstractFloat
@show eps(Float64)
@show eps(BigFloat);

    Float64 <: AbstractFloat = true
BigFloat <: AbstractFloat = true
eps(Float64) = 2.220446049250313e-16
eps(BigFloat) = 1.727233711818889250772703725600799142232000072887256277004740694033718360632485e-77

The isless function also has multiple methods.

First let’s try with integers

In [32]: @which isless(1, 2)

Out[32]: isless(x::Real, y::Real) in Base at operators.jl:346

As we saw previously, the Real data type is an abstract type, and encompasses both floats and integers.

If we go to the provided link in the source, we see the entirety of the function is

isless(x::Real, y::Real) = x<y

That is, for any values where typeof(x) <: Real and typeof(y) <: Real, the definition relies on <.

We know that < is defined for the types because it is part of the informal interface for the Real abstract type.

Note that this is not defined for Number because not all Number types have the < ordering operator defined (e.g. Complex).

In order to generate fast code, the implementation details may define specialized versions of these operations.

In [33]: isless(1.0, 2.0)  # applied to two floats
 @which isless(1.0, 2.0)

Out[33]: isless(x::Float64, y::Float64) in Base at float.jl:465

Note that the reason Float64 <: Real calls this implementation rather than the one given above, is that Float64 <: Real, and Julia chooses the most specialized implementation for each function.

The specialized implementations are often more subtle than you may realize due to floating point arithmetic, underflow, etc.
7.7 Functions, and Function-Like Types

Another common example of the separation between data structures and algorithms is the use of functions.

Syntactically, a univariate “function” is any $f$ that can call an argument $x$ as $f(x)$.

For example, we can use a standard function

In [34]: using QuadGK
   f(x) = x^2
  @show quadgk(f, 0.0, 1.0) # integral

   function plotfunctions(f)
       intf(x) = quadgk(f, 0.0, x)[1] # $\int_0^x f(x) \, dx$
       x = 0:0.1:1.0
       f_x = f(x)
       plot(x, f_x, label="f")
       plot!(x, intf(x), label="int_f")
   end
   plotfunctions(f)  # call with our f

   quadgk(f, 0.0, 1.0) = (0.3333333333333333, 5.551115123125783e-17)

Out[34]:

Of course, univariate polynomials are another type of univariate function

In [35]: using Polynomials
   p = Polynomial([2, -5, 2], :x)  # :x just gives a symbol for display
Similarly, the result of interpolating data is also a function

In [36]: using Interpolations

\[ x = 0.0:0.2:1.0 \]
\[ f(x) = x^2 \]
\[ f\_int = \text{LinearInterpolation}(x, f(x)) \]  # interpolates the coarse grid
\[ @show f\_int(1.0) \]  # call like a function

plotfunctions(f\_int)  # same generic function

\[ f\_int(1.0) = 1.0 \]
Note that the same generic plotfunctions could use any variable passed to it that “looks” like a function, i.e., can call \( f(x) \).

This approach to design with types – generic, but without any specific type declarations – is called duck typing.

If you need to make an existing type callable, see Function Like Objects.

### 7.8 Limitations of Dispatching on Abstract Types

You will notice that types in Julia represent a tree with \texttt{Any} at the root.

The tree structure has worked well for the above examples, but it doesn’t allow us to associate multiple categorizations of types.

For example, a semi-group type would be useful for a writing generic code (e.g. continuous-time solutions for ODEs and matrix-free methods), but cannot be implemented rigorously since the \texttt{Matrix} type is a semi-group as well as an \texttt{AbstractArray}, but not all semi-groups are \texttt{AbstractArray}s.

The main way to implement this in a generic language is with a design approach called “traits”.

- See the original discussion and an example of a package to facilitate the pattern.
- A complete description of the traits pattern as the natural evolution of Multiple Dispatch is given in this blog post.
### 7.9 Exercises

#### 7.9.1 Exercise 1a

In a previous exercise, we discussed the **trapezoidal rule** for numerical integration. To summarize, the vector

\[
\int_{\underline{x}}^{\bar{x}} f(x) \, dx \approx \omega \cdot \bar{f}
\]

where \( \bar{f} \equiv \begin{bmatrix} f(x_1) & \ldots & f(x_N) \end{bmatrix} \in \mathbb{R}^N \) and, for a uniform grid spacing of \( \Delta \),

\[
\omega \equiv \Delta \begin{bmatrix} \frac{1}{2} & 1 & \ldots & 1 & \frac{1}{2} \end{bmatrix} \in \mathbb{R}^N
\]

The quadrature rule can be implemented easily as

**In** [37]:

```plaintext
using LinearAlgebra

function trap_weights(x)
    return step(x) * [0.5; ones(length(x) - 2); 0.5]
end

x = range(0.0, 1.0, length = 100)
ω = trap_weights(x)
f(x) = x^2
dot(f.(x), ω)
```

**Out**[37]: 

0.3333503384008434

However, in this case the creation of the \( \omega \) temporary is inefficient as there are no reasons to allocate an entire vector just to iterate through it with the `dot`. Instead, create an iterable by following the interface definition for `Iteration`, and implement the modified `trap_weights` and integration.

**Hint**: create a type such as

**In** [38]:

```plaintext
struct UniformTrapezoidal
    count::Int
    Δ::Float64
end
```

and then implement the function `Base.iterate(S::UniformTrapezoidal, state=1)`.

#### 7.9.2 Exercise 1b (Advanced)

Make the `UniformTrapezoidal` type operate as an array with interface definition for `AbstractArray`. With this, you should be able it go \( \omega[2] \) or `length(ω)` to access the quadrature weights.

#### 7.9.3 Exercise 2 (Advanced)

Implement the same features as Exercise 1a and 1b, but for the **non-uniform trapezoidal rule**.
Chapter 8

General Purpose Packages

8.1 Contents

- Overview 8.2
- Numerical Integration 8.3
- Interpolation 8.4
- Linear Algebra 8.5
- General Tools 8.6

8.2 Overview

Julia has both a large number of useful, well written libraries and many incomplete poorly maintained proofs of concept.

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.

Also see data and statistical packages and optimization, solver, and related packages for more domain specific packages.

8.2.1 Setup

In [1]: using InstantiateFromURL
   # optionally add arguments to force installation: instantiate = true,[]
   precompile = true
   github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

In [2]: using LinearAlgebra, Statistics
   using QuantEcon, QuadGK, FastGaussQuadrature, Distributions, Expectations
   using Interpolations, Plots, LaTeXStrings, ProgressMeter
8.3 Numerical Integration

Many applications require directly calculating a numerical derivative and calculating expectations.

8.3.1 Adaptive Quadrature

A high accuracy solution for calculating numerical integrals is \texttt{QuadGK}.

```
In [3]: using QuadGK
   @show value, tol = quadgk(cos, -2π, 2π);

     (value, tol) = quadgk(cos, -2π, 2π) = (-1.5474478810961125e-14, 5.7846997329025695e-24)
```

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.

8.3.2 Gaussian Quadrature

Alternatively, many integrals can be done efficiently with (non-adaptive) \texttt{Gaussian quadrature}.

For example, using \texttt{FastGaussQuadrature.jl}

```
In [4]: using FastGaussQuadrature
   x, w = gausslegendre(100_000);  # i.e. find 100,000 nodes

   # integrates f(x) = x^2 from -1 to 1
   f(x) = x^2
   @show w √ f.(x);  # calculate integral

   w √ f.(x) = 0.6666666666666667
```

The only problem with the \texttt{FastGaussQuadrature} package is that you will need to deal with affine transformations to the non-default domains yourself.
Alternatively, \texttt{QuantEcon.jl} has routines for Gaussian quadrature that translate the domains.

```
In [5]: using QuantEcon

   x, w = qnwlege(65, -2π, 2π);
   @show w √ cos.(x);  # i.e. on [-2π, 2π] domain

   w √ cos.(x) = -3.0064051806277455e-15
```
8.3.3 Expectations

If the calculations of the numerical integral is simply for calculating mathematical expectations of a particular distribution, then Expectations.jl provides a convenient interface.

Under the hood, it is finding the appropriate Gaussian quadrature scheme for the distribution using FastGaussQuadrature.

In [6]: using Distributions, Expectations
    dist = Normal()
    E = expectation(dist)
    f(x) = x
   @show E(f) #i.e. identity

    # Or using as a linear operator
    f(x) = x^2
    x = nodes(E)
    w = weights(E)
    E * f.(x) == f.(x) ⊙ w

    E(f) = -6.991310601309959e-18

Out[6]: true

8.4 Interpolation

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

The package we usually turn to for this purpose is Interpolations.jl.

There are a variety of options, but we will only demonstrate the convenient notations.

8.4.1 Univariate with a Regular Grid

Let’s start with the univariate case.

We begin by creating some data points, using a sine function

In [7]: using Interpolations
    using Plots
    gr(fmt=:png);  
    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 = "sin function")
    scatter!(x, y, label = "sampled data", markersize = 4)

Out[7]:
To implement linear and cubic spline interpolation

In [8]: li = LinearInterpolation(x, y)
li_spline = CubicSplineInterpolation(x, y)
@show li(0.3) # evaluate at a single point

scatter(x, y, label = "sampled data", markersize = 4)
plot!(xf, li(xf), label = "linear")
plot!(xf, li_spline(xf), label = "spline")

li(0.3) = 0.25244129544236954

Out[8]: 
8.4.2 Univariate with Irregular Grid

In the above, the `LinearInterpolation` function uses a specialized function for regular grids since `x` is a `Range` type.

For an arbitrary, irregular grid

```
In [9]: x = log.(range(1, exp(4), length = 10)) .+ 1  # uneven grid
   y = log.(x) # corresponding y points

   interp = LinearInterpolation(x, y)

   xf = log.(range(1, exp(4), length = 100)) .+ 1  # finer grid

   plot(xf, interp.(xf), label = "linear")
   scatter!(x, y, label = "sampled data", markersize = 4, size = (800, 400))
```

Out[9]:
At this point, *Interpolations.jl* does not have support for cubic splines with irregular grids, but there are plenty of other packages that do (e.g. *Dierckx.jl* and *GridInterpolations.jl*).

### 8.4.3 Multivariate Interpolation

Interpolating a regular multivariate function uses the same function

In [10]: \[ f(x, y) = \log(x+y) \]
\[
\begin{align*}
xs &= 1:0.2:5 \\
ys &= 2:0.1:5 \\
A &= [f(x, y) \text{ for } x \text{ in } xs, y \text{ in } ys] \\
\end{align*}
\]

\[
\begin{align*}
# \text{ linear interpolation} \\
\text{interp\_linear} &= \text{LinearInterpolation}((xs, ys), A) \\
@\text{show} \ \text{interp\_linear}(3, 2) \ # \text{exactly } \log(3 + 2) \\
@\text{show} \ \text{interp\_linear}(3.1, 2.1) \ # \text{approximately } \log(3.1 + 2.1) \\
\end{align*}
\]

\[
\begin{align*}
# \text{ cubic spline interpolation} \\
\text{interp\_cubic} &= \text{CubicSplineInterpolation}((xs, ys), A) \\
@\text{show} \ \text{interp\_cubic}(3, 2) \ # \text{exactly } \log(3 + 2) \\
@\text{show} \ \text{interp\_cubic}(3.1, 2.1) \ # \text{approximately } \log(3.1 + 2.1); \\
\end{align*}
\]

\[
\begin{align*}
\text{interp\_linear}(3, 2) &= 1.6094379124341003 \\
\text{interp\_linear}(3.1, 2.1) &= 1.6484736801441782 \\
\text{interp\_cubic}(3, 2) &= 1.6094379124341003 \\
\text{interp\_cubic}(3.1, 2.1) &= 1.6486586594237707 \\
\end{align*}
\]

See *Interpolations.jl documentation* for more details on options and settings.
8.5 Linear Algebra

8.5.1 Standard Library

The standard library contains many useful routines for linear algebra, in addition to standard functions such as \texttt{det()}, \texttt{inv()}, \texttt{factorize()}, etc.

Routines are available for

- Cholesky factorization
- LU decomposition
- Singular value decomposition,
- Schur factorization, etc.

See here for further details.

8.6 General Tools

8.6.1 \LaTeXStrings.jl

When you need to properly escape \LaTeX code (e.g. for equation labels), use \LaTeXStrings.jl.

In [11]: using \LaTeXStrings
   \LaTeX"an equation: $1 + \alpha^2$"

Out[11]: an equation: $1 + \alpha^2$

8.6.2 ProgressMeter.jl

For long-running operations, you can use the \ProgressMeter.jl package.

To use the package, you simply put a macro in front of \texttt{for} loops, etc.

From the documentation

In [12]: using ProgressMeter

\@showprogress 1 "Computing..." \texttt{for i in 1:50}
   \texttt{sleep(0.1)} \# some computation...
   \texttt{end}

Computing...100%|██████████████████████████████████████| Time: 0:00:05
Chapter 9

Data and Statistics Packages

9.1 Contents

- Overview 9.2
- DataFrames 9.3
- Statistics and Econometrics 9.4

9.2 Overview

This lecture explores some of the key packages for working with data and doing statistics in Julia.

In particular, we will examine the DataFrame object in detail (i.e., construction, manipulation, querying, visualization, and nuances like missing data).

While Julia is not an ideal language for pure cookie-cutter statistical analysis, it has many useful packages to provide those tools as part of a more general solution.

This list is not exhaustive, and others can be found in organizations such as JuliaStats, JuliaData, and QueryVerse.

9.2.1 Setup

In [1]: using InstantiateFromURL
   # optionally add arguments to force installation: instantiate = true,
   "precompile = true"
   github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

In [2]: using LinearAlgebra, Statistics
   using DataFrames, RDatasets, DataFramesMeta, CategoricalArrays, Query,
   "VegaLite"
   using GLM

9.3 DataFrames

A useful package for working with data is DataFrames.jl.
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 (of the same type).

This is analogous to a data.frame in R, a DataFrame in Pandas (Python) or, more loosely, a spreadsheet in Excel.

There are a few different ways to create a DataFrame.

### 9.3.1 Constructing and Accessing a DataFrame

The first is to set up columns and construct a dataframe by assigning names using DataFrames, RDatasets

```
# RDatasets provides good standard data examples from R

# note use of missing
commodities = ["crude", "gas", "gold", "silver"]
last_price = [4.2, 11.3, 12.1, missing]
df = DataFrame(commod = commodities, price = last_price)
```

```
\begin{tabular}{r|cc}
  & commod & price \\
  \hline
  1 & crude & 4.2 \\
  2 & gas & 11.3 \\
  3 & gold & 12.1 \\
  4 & silver & \textbf{missing} \\
\end{tabular}
```

Columns of the DataFrame can be accessed by name using `df.col`, as below

```
In [4]: df.price
```

```
Out[4]: 4-element Array{Union{Missing, Float64},1}:
    4.2
    11.3
    12.1
    missing
```

Note that the type of this array has values `Union{Missing, Float64}` since it was created with a `missing` value.

```
In [5]: df.commod
```

```
Out[5]: 4-element Array{String,1}:
    "crude"
    "gas"
    "gold"
    "silver"
```
The `DataFrames.jl` package provides a number of methods for acting on DataFrame's, such as `describe`.

In [6]: `DataFrames.describe(df)`

Out[6]:
```
\begin{tabular}{r|cccccccc}
& variable & mean & min & median & max & nunique & nmissing & eltype \\
\hline
& Symbol & …Union & Any & …Union & Any & …Union & …Union & Type \\
\hline
1 & commod & & crude & & silver & 4 & & String \\
2 & price & 9.2 & 4.2 & 11.3 & 12.1 & & 1 & Union\{Missing, Float64\} \\
\end{tabular}
```

While often data will be generated all at once, or read from a file, you can add to a DataFrame by providing the key parameters.

In [7]: `nt = (commod = "nickel", price= 5.1)
push!(df, nt)`

Out[7]:
```
\begin{tabular}{r|cc}
& commod & price \\
\hline
& String & Float64? \\
\hline
1 & crude & 4.2 \\
2 & gas & 11.3 \\
3 & gold & 12.1 \\
4 & silver & \textbf{missing} \\
5 & nickel & 5.1 \\
\end{tabular}
```

Named tuples can also be used to construct a DataFrame, and have it properly deduce all types.

In [8]: `nt = (t = 1, col1 = 3.0)
df2 = DataFrame([nt])
push!(df2, (t=2, col1 = 4.0))`

Out[8]:
```
\begin{tabular}{r|cc}
& t & col1 \\
\hline
& Int64 & Float64 \\
\hline
1 & 1 & 3.0 \\
2 & 2 & 4.0 \\
\end{tabular}
```

In order to modify a column, access the mutating version by the symbol `df[!, :col]`.

In [9]: `df[!, :price]`

Out[9]: 5-element Array{Union{Missing, Float64},1}:
```
Which allows modifications, like other mutating functions in Julia.

In [10]: df[:, :price] *= 2.0  # double prices

Out[10]: 5-element Array{Union{Missing, Float64},1}:
          8.4
          22.6
          24.2
         missing
          10.2

As discussed in the next section, note that the fundamental types, is propagated, i.e. 
missing * 2 === missing.

9.3.2 Working with Missing

As we discussed in fundamental types, the semantics of missing are that mathematical operations will not silently ignore it.

In order to allow missing in a column, you can create/load the DataFrame from a source with missing’s, or call allowmissing! on a column.

In [11]: allowmissing!(df2, :col1)  # necessary to add in a for col1
   push!(df2, (t=3, col1 = missing))
   push!(df2, (t=4, col1 = 5.1))

Out[11]: \begin{tabular}{r|cc}
         & t & col1\
         \hline
         1 & 1 & 3.0 \\
         2 & 2 & 4.0 \\
         3 & 3 & \textbf{missing} \\
         4 & 4 & 5.1 \\
\end{tabular}

We can see the propagation of missing to caller functions, as well as a way to efficiently calculate with non-missing data.

In [12]:@show mean(df2.col1)
   @show mean(skipmissing(df2.col1))

     mean(df2.col1) = missing
     mean(skipmissing(df2.col1)) = 4.033333333333333

Out[12]: 4.0333333333333333
9.3. DATAFRAMES

And to replace the missing

In [13]: df2.col1 .= coalesce.(df2.col1, 0.0)  # replace all missing with 0.0

Out[13]: 4-element Array{Union{Missing, Float64},1}:
3.0
4.0
0.0
5.1

9.3.3 Manipulating and Transforming DataFrames

One way to do an additional calculation with a DataFrame is to use the @transform macro from DataFramesMeta.jl.

In [14]: using DataFramesMeta
f(x) = x^2
df2 = @transform(df2, col2 = f(col1))

Out[14]: \begin{tabular}{r|ccc}
  & t & col1 & col2 \\
  & Int64 & Float64? & Float64 \\
 1 & 1 & 3.0 & 9.0 \\
2 & 2 & 4.0 & 16.0 \\
3 & 3 & 0.0 & 0.0 \\
4 & 4 & 5.1 & 26.01 \\
\end{tabular}

9.3.4 Categorical Data

For data that is categorical

In [15]: using CategoricalArrays
id = [1, 2, 3, 4]
y = ["old", "young", "young", "old"]
y = CategoricalArray(y)
df = DataFrame(id=id, y=y)

Out[15]: \begin{tabular}{r|cc}
  & id & y \\
  & Int64 & ...Cat \\
 1 & 1 & old \\
2 & 2 & young \\
3 & 3 & young \\
4 & 4 & old \\
\end{tabular}

In [16]: levels(df.y)
Out[16]: 2-element Array{String,1}:
  "old"
  "young"

9.3.5 Visualization, Querying, and Plots

The DataFrame (and similar types that fulfill a standard generic interface) can fit into a variety of packages.

One set of them is the QueryVerse.

Note: The QueryVerse, in the same spirit as R’s tidyverse, makes heavy use of the pipeline syntax `|>`.  

In [17]: \[x = 3.0\]
   \[f(x) = x^2\]
   \[g(x) = \log(x)\]

   @show g(f(x))
   @show x |> f |> g; # pipes nest function calls

 \[g(f(x)) = 2.1972245773362196\]
 \[(x |> f) |> g = 2.1972245773362196\]

To give an example directly from the source of the LINQ inspired Query.jl

In [18]: using Query

   df = DataFrame(name=\["John", "Sally", "Kirk"\], age=\[23., 42., 59.\],
   children=\[3, 5, 2\])

   x = @from i in df begin
      @where i.age > 50
      @select \{i.name, i.children\}
      @collect DataFrame
   end

Out[18]: \[
\begin{tabular}{r|cc}
  & name & children \\
  \hline
  & String & Int64 \\
  \hline
  1 & Kirk & 2 \\
\end{tabular}
\]

While it is possible to just use the Plots.jl library, there may be better options for displaying tabular data – such as VegaLite.jl.

In [19]: using RDatasets, VegaLite

   iris = dataset("datasets", "iris")

   iris |> @vlplot(
      :point,
9.4 Statistics and Econometrics

While Julia is not intended as a replacement for R, Stata, and similar specialty languages, it has a growing number of packages aimed at statistics and econometrics.

Many of the packages live in the JuliaStats organization.

A few to point out

- **StatsBase** has basic statistical functions such as geometric and harmonic means, autocorrelations, robust statistics, etc.
- **StatsFuns** has a variety of mathematical functions and constants such as pdf and cdf of many distributions, softmax, etc.

### 9.4.1 General Linear Models

To run linear regressions and similar statistics, use the `GLM` package.

```plaintext
In [20]: using GLM

x = randn(100)
y = 0.9 .* x + 0.5 .* rand(100)
df = DataFrame(x=x, y=y)
ols = lm(@formula(y ~ x), df) # R-style notation

Out[20]: StatsModels.TableRegressionModel{LinearModel{GLM.LmResp{Array{Float64,1}},GLM.DensePredC hol{Float64,Cholesky{Float64,Array{Float64,2}}}},Array{Float64,2}}
y ~ 1 + x

Coefficients:

|                | Estimate | Std. Error | t value | Pr(>|t|) | Lower 95% | Upper 95% |
|----------------|----------|------------|---------|----------|-----------|-----------|
| (Intercept)    | 0.228452 | 0.01451    | 15.7445 | <1e-27   | 0.199657  | 0.257247  |
| x              | 0.910643 | 0.0130831  | 69.6044 | <1e-84   | 0.88468   | 0.936606  |
```

To display the results in a useful tables for LaTeX and the REPL, use `RegressionTables` for output similar to the Stata package `esttab` and the R package `stargazer`.
In [21]: using RegressionTables
    regtable(ols)
    # regtable(ols, renderSettings = latexOutput()) # for LaTex output

----------------------

y

(1)

----------------------

(Intercept) 0.228***
     (0.015)
    x 0.911***
     (0.013)

Estimator OLS

----------------------

N  100
R2 0.980

9.4.2 Fixed Effects

While Julia may be overkill for estimating a simple linear regression, fixed-effects estimation with dummies for multiple variables are much more computationally intensive.

For a 2-way fixed-effect, taking the example directly from the documentation using cigarette consumption data

In [22]: using FixedEffectModels
cigar = dataset("plm", "Cigar")
cigar.StateCategorical = categorical(cigar.State)
cigar.YearCategorical = categorical(cigar.Year)
fixedeffectresults = reg(cigar, @formula(Sales ~ NDI +]
   .fe(StateCategorical) +
   .fe(YearCategorical)), weights = :Pop, Vcov.cluster(:State))
regtable(fixedeffectresults)

----------------------

Sales

(1)

----------------------

NDI -0.005***
     (0.001)

StateCategorical Yes
YearCategorical Yes

Estimator OLS

----------------------

N 1,380
R2 0.803
Chapter 10

Solvers, Optimizers, and Automatic Differentiation

10.1 Contents

- Overview 10.2
- Introduction to Differentiable Programming 10.3
- Optimization 10.4
- Systems of Equations and Least Squares 10.5
- LeastSquaresOptim.jl 10.6
- Additional Notes 10.7
- Exercises 10.8

10.2 Overview

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

10.2.1 Setup

In [1]: using InstantiateFromURL
   # optionally add arguments to force installation: instantiate = true,
   # precompile = true
   github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

In [2]: using LinearAlgebra, Statistics
   using ForwardDiff, Zygote, Optim, JuMP, Ipopt, BlackBoxOptim, Roots,
   NLsolve, LeastSquaresOptim
   using Optim: converged, maximum, maximizer, minimizer, iterations #some
   #extra functions
10.3 Introduction to Differentiable Programming

The promise of differentiable programming is that we can move towards taking the derivatives of almost arbitrarily complicated computer programs, rather than simply thinking about the derivatives of mathematical functions. Differentiable programming is the natural evolution of automatic differentiation (AD, sometimes called algorithmic differentiation).

Stepping back, there are three ways to calculate the gradient or Jacobian

- **Analytic derivatives / Symbolic differentiation**
  - You can sometimes calculate the derivative on pen-and-paper, and potentially simplify the expression.
  - In effect, repeated applications of the chain rule, product rule, etc.
  - It is sometimes, though not always, the most accurate and fastest option if there are algebraic simplifications.
  - Sometimes symbolic integration on the computer a good solution, if the package can handle your functions. Doing algebra by hand is tedious and error-prone, but is sometimes invaluable.

- **Finite differences**
  - Evaluate the function at least $N + 1$ times to get the gradient – Jacobians are even worse.
  - Large $\Delta$ is numerically stable but inaccurate, too small of $\Delta$ is numerically unstable but more accurate.
  - Choosing the $\Delta$ is hard, so use packages such as DiffEqDiffTools.jl.
  - If a function is $\mathbb{R}^N \to \mathbb{R}$ for a large $N$, this requires $O(N)$ function evaluations.

- **Automatic Differentiation**
  - The same as analytic/symbolic differentiation, but where the chain rule is calculated numerically rather than symbolically.
  - Just as with analytic derivatives, can establish rules for the derivatives of individual functions (e.g. $d(sin(x))$ to $cos(x)dx$) for intrinsic derivatives.

AD has two basic approaches, which are variations on the order of evaluating the chain rule: reverse and forward mode (although mixed mode is possible).

1. If a function is $\mathbb{R}^N \to \mathbb{R}$, then **reverse-mode** AD can find the gradient in $O(1)$ sweep (where a “sweep” is $O(1)$ function evaluations).

2. If a function is $\mathbb{R} \to \mathbb{R}^N$, then **forward-mode** AD can find the jacobian in $O(1)$ sweeps.

We will explore two types of automatic differentiation in Julia (and discuss a few packages which implement them). For both, remember the chain rule

$$\frac{dy}{dx} = \frac{dy}{dw} \cdot \frac{dw}{dx}$$

Forward-mode starts the calculation from the left with $\frac{dy}{dx}$ first, which then calculates the product with $\frac{dw}{dx}$. On the other hand, reverse mode starts on the right hand side with $\frac{dw}{dx}$ and works backwards.
Take an example a function with fundamental operations and known analytical derivatives

\[ f(x_1, x_2) = x_1 x_2 + \sin(x_1) \]

And rewrite this as a function which contains a sequence of simple operations and temporaries.

In [3]: function f(x_1, x_2)
    w_1 = x_1
    w_2 = x_2
    w_3 = w_1 * w_2
    w_4 = sin(w_1)
    w_5 = w_3 + w_4
    return w_5
end

Out[3]: f (generic function with 1 method)

Here we can identify all of the underlying functions (\(*, \sin, +\)), and see if each has an intrinsic derivative. While these are obvious, with Julia we could come up with all sorts of differentiation rules for arbitrarily complicated combinations and compositions of intrinsic operations. In fact, there is even a package for registering more.

10.3.1 Forward-Mode Automatic Differentiation

In forward-mode AD, you first fix the variable you are interested in (called “seeding”), and then evaluate the chain rule in left-to-right order.

For example, with our \( f(x_1, f_2) \) example above, if we wanted to calculate the derivative with respect to \( x_1 \) then we can seed the setup accordingly. \( \frac{\partial w_1}{\partial x_1} = 1 \) since we are taking the derivative of it, while \( \frac{\partial w_1}{\partial x_1} = 0 \).

Following through with these, redo all of the calculations for the derivative in parallel with the function itself.

<table>
<thead>
<tr>
<th>( f(x_1, x_2) )</th>
<th>( \frac{\partial f(x_1, x_2)}{\partial x_1} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( w_1 = x_1 )</td>
<td>( \frac{\partial w_1}{\partial x_1} = 1 ) (seed)</td>
</tr>
<tr>
<td>( w_2 = x_2 )</td>
<td>( \frac{\partial w_2}{\partial x_1} = 0 ) (seed)</td>
</tr>
<tr>
<td>( w_3 = w_1 \cdot w_2 )</td>
<td>( \frac{\partial w_3}{\partial x_1} = w_2 \cdot \frac{\partial w_1}{\partial x_1} + w_1 \cdot \frac{\partial w_2}{\partial x_1} )</td>
</tr>
<tr>
<td>( w_4 = \sin w_1 )</td>
<td>( \frac{\partial w_4}{\partial x_1} = \cos w_1 \cdot \frac{\partial w_1}{\partial x_1} )</td>
</tr>
<tr>
<td>( w_5 = w_3 + w_4 )</td>
<td>( \frac{\partial w_5}{\partial x_1} = \frac{\partial w_3}{\partial x_1} + \frac{\partial w_4}{\partial x_1} )</td>
</tr>
</tbody>
</table>

Since these two could be done at the same time, we say there is “one pass” required for this calculation.

Generalizing a little, if the function was vector-valued, then that single pass would get the entire row of the Jacobian in that single pass. Hence for a \( R^N \rightarrow R^M \) function, requires \( N \) passes to get a dense Jacobian using forward-mode AD.

How can you implement forward-mode AD? It turns out to be fairly easy with a generic programming language to make a simple example (while the devil is in the details for a high-performance implementation).
10.3.2 Forward-Mode with Dual Numbers

One way to implement forward-mode AD is to use dual numbers. Instead of working with just a real number, e.g. \( x \), we will augment each with an infinitesimal \( \epsilon \) and use \( x + \epsilon \).

From Taylor’s theorem,

\[
f(x + \epsilon) = f(x) + f'(x)\epsilon + O(\epsilon^2)
\]

where we will define the infinitesimal such that \( \epsilon^2 = 0 \).

With this definition, we can write a general rule for differentiation of \( g(x, y) \) as the chain rule for the total derivative

\[
g(x + \epsilon, y + \epsilon) = g(x, y) + (\partial_x g(x, y) + \partial_y g(x, y))\epsilon
\]

But, note that if we keep track of the constant in front of the \( \epsilon \) terms (e.g. a \( x' \) and \( y' \))

\[
g(x + x'\epsilon, y + y'\epsilon) = g(x, y) + (\partial_x g(x, y)x' + \partial_y g(x, y)y')\epsilon
\]

This is simply the chain rule. A few more examples

\[
(x + x'\epsilon) + (y + y'\epsilon) = (x + y) + (x' + y')\epsilon
\]
\[
(x + x'\epsilon) \times (y + y'\epsilon) = (xy) + (x'y + y'x)\epsilon
\]
\[
\exp(x + x'\epsilon) = \exp(x) + (x' \exp(x))\epsilon
\]

Using the generic programming in Julia, it is easy to define a new dual number type which can encapsulate the pair \( (x, x') \) and provide a definitions for all of the basic operations. Each definition then has the chain-rule built into it.

With this approach, the “seed” process is simple the creation of the \( \epsilon \) for the underlying variable.

So if we have the function \( f(x_1, x_2) \) and we wanted to find the derivative \( \partial_{x_1} f(3.8, 6.9) \) then we would seed them with the dual numbers \( x_1 \to (3.8, 1) \) and \( x_2 \to (6.9, 0) \).

If you then follow all of the same scalar operations above with a seeded dual number, it will calculate both the function value and the derivative in a single “sweep” and without modifying any of your (generic) code.

10.3.3 ForwardDiff.jl

Dual-numbers are at the heart of one of the AD packages we have already seen.

In [4]: using ForwardDiff

\[
h(x) = \sin(x[1]) + x[1] \times x[2] + \sinh(x[1] \times x[2]) \quad \text{# multivariate.}
\]
\[
x = [1.4 \ 2.2]
\]

@show ForwardDiff.gradient(h,x) \ # use AD, seeds from \( x \)

# Or, can use complicated functions of many variables

\[
f(x) = \text{sum}(\sin, x) + \text{prod}(\tan, x) \times \text{sum}(\text{sqrt}, x)
\]
10.3. INTRODUCTION TO DIFFERENTIABLE PROGRAMMING

\[ g = (x) \rightarrow \text{ForwardDiff.gradient}(f, x); \quad \# g() \text{ is now the gradient} \]
\[ g(\text{rand}(5)) \quad \# \text{gradient at a random point} \]
\[ \text{ForwardDiff.hessian}(f, x') \quad \# \text{or the hessian} \]

```
ForwardDiff.gradient(h, x) = [26.354764961030977 16.663053156992284]
```

```
Out[4]: 5-element Array{Float64,1}:
  1.1951182078800449
  1.653216543645516
  1.214581567366202
  1.394366855783785
  1.1889756683992878
```

We can even auto-differentiate complicated functions with embedded iterations.

```
function squareroot(x)
    \# pretending we don't know sqrt()
    z = copy(x) \# Initial starting point for Newton's method
    while abs(z^z - x) > 1e-13
        z = z - (z^z-x)/(2z)
    end
    return z
end
squareroot(2.0)
```

```
Out[5]: 1.4142135623730951
```

```
In [6]: using ForwardDiff
dsqrt(x) = ForwardDiff.derivative(squareroot, x)
dsqrt(2.0)
```

```
Out[6]: 0.35355339059327373
```

10.3.4 Zygote.jl

Unlike forward-mode auto-differentiation, reverse-mode is very difficult to implement efficiently, and there are many variations on the best approach.

Many reverse-mode packages are connected to machine-learning packages, since the efficient gradients of \( \mathbb{R}^N \rightarrow \mathbb{R} \) loss functions are necessary for the gradient descent optimization algorithms used in machine learning.

One recent package is Zygote.jl, which is used in the Flux.jl framework.

```
In [7]: using Zygote
    h(x, y) = 3x^2 + 2x + 1 + y^x - y
    gradient(h, 3.0, 5.0)
```

```
Out[7]: (25.0, 2.0)
```

Here we see that Zygote has a gradient function as the interface, which returns a tuple.

You could create this as an operator if you wanted to.
In [8]: $D(f) = x -> \text{gradient}(f, x)[1]$  # returns first in tuple

$D_{\sin} = D(\sin)$

$D_{\sin}(4.0)$

Out[8]: -0.6536436208636119

For functions of one (Julia) variable, we can find the by simply using the `' after a function name

In [9]: using Statistics

$p(x) = \text{mean(abs, x)}$

$p'([1.0, 3.0, -2.0])$

Out[9]: 3-element Array{Float64,1}:
        0.3333333333333333
        0.3333333333333333
       -0.3333333333333333

Or, using the complicated iterative function we defined for the squareroot,

In [10]: squareroot'(2.0)

Out[10]: 0.3535533905932737

Zygote supports combinations of vectors and scalars as the function parameters.

In [11]: $h(x, n) = (\text{sum}(x.^n))^{(1/n)}$

$\text{gradient}(h, [1.0, 4.0, 6.0], 2.0)$

Out[11]: ([0.13736056394868904, 0.5494422557947561, 0.8241633836921343], -1.272553130925444 + 0.0im)

The gradients can be very high dimensional. For example, to do a simple nonlinear optimization problem with 1 million dimensions, solved in a few seconds.

In [12]: using Optim, LinearAlgebra

$N = 1000000$

$y = \text{rand}(N)$

$\lambda = 0.01$

$obj(x) = \text{sum}((x - y).^2) + \lambda*\text{norm}(x)$

$x_{iv} = \text{rand}(N)$

function $g!(G, x)$

$G .= obj'(x)$

end

$results = \text{optimize}(obj, g!, x_{iv}, \text{LBFGS}())$  # or ConjugateGradient()

println("minimum = ", results.minimum) with in "
"$results.\text{iterations}$ iterations")
Caution: while Zygote is the most exciting reverse-mode AD implementation in Julia, it has many rough edges.

- If you write a function, take its gradient, and then modify the function, you need to call `Zygote.refresh()` or else the gradient will be out of sync. This may not apply for Julia 1.3+.
- It provides no features for getting Jacobians, so you would have to ask for each row of the Jacobian separately. That said, you probably want to use `ForwardDiff.jl` for Jacobians if the dimension of the output is similar to the dimension of the input.
- You cannot, in the current release, use mutating functions (e.g. modify a value in an array/etc.) although that feature is in progress.
- Compiling can be very slow for complicated functions.

### 10.4 Optimization

There are a large number of packages intended to be used for optimization in Julia.

Part of the reason for the diversity of options is that Julia makes it possible to efficiently implement a large number of variations on optimization routines.

The other reason is that different types of optimization problems require different algorithms.

#### 10.4.1 Optim.jl

A good pure-Julia solution for the (unconstrained or box-bounded) optimization of univariate and multivariate function is the `Optim.jl` package.

By default, the algorithms in `Optim.jl` target minimization rather than maximization, so if a function is called `optimize` it will mean minimization.

**Univariate Functions on Bounded Intervals**

Univariate optimization defaults to a robust hybrid optimization routine called Brent’s method.

In [13]: using Optim
    using Optim: converged, maximum, maximizer, minimizer, iterations #some extra functions
    result = optimize(x-> x^2, -2.0, 1.0)

Out[13]: Results of Optimization Algorithm
- Algorithm: Brent's Method
- Search Interval: [-2.000000, 1.000000]
- Minimizer: 0.000000e+00
- Minimum: 0.000000e+00
- Iterations: 5
- Convergence: max(|x - x_upper|, |x - x_lower|) <= 2*(1.5e-08*|x|+2.2e-16): true
- Objective Function Calls: 6
Always check if the results converged, and throw errors otherwise

```python
In [14]: converged(result) || error("Failed to converge in $(iterations(result)) iterations")
xmin = result.minimizer
result.minimum

Out[14]: 0.0
```

The first line is a logical OR between `converged(result)` and `error("...")`. If the convergence check passes, the logical sentence is true, and it will proceed to the next line; if not, it will throw the error.

Or to maximize

```python
In [15]: f(x) = -x^2
result = maximize(f, -2.0, 1.0)
converged(result) || error("Failed to converge in $(iterations(result)) iterations")
xmin = maximizer(result)
fmax = maximum(result)

Out[15]: -0.0
```

**Note:** Notice that we call `optimize` results using `result.minimizer`, and `maximize` results using `maximizer(result)`.

### Unconstrained Multivariate Optimization

There are a variety of algorithms and options for multivariate optimization.

From the documentation, the simplest version is

```python
In [16]: f(x) = (1.0 - x[1])^2 + 100.0 * (x[2] - x[1]^2)^2
x_iv = [0.0, 0.0]
results = optimize(f, x_iv) # i.e. optimize(f, x_iv, NelderMead())

Out[16]: * Status: success

* Candidate solution
  Minimizer: [1.00e+00, 1.00e+00]
  Minimum: 3.525527e-09

* Found with
  Algorithm: Nelder-Mead
  Initial Point: [0.00e+00, 0.00e+00]

* Convergence measures
  \( \sqrt{\sum(y_i - \bar{y})^2}/n \leq 1.0e-08 \)

* Work counters
  Seconds run: 0 (vs limit Inf)
  Iterations: 60
  f(x) calls: 118
```
The default algorithm in **NelderMead**, which is derivative-free and hence requires many function evaluations.

To change the algorithm type to **L-BFGS**

In [17]: results = optimize(f, x_iv, LBFGS())

```
println("minimum = \$(results.minimum) with argmin = \$(results.minimizer)\n
minimum = 5.3784046148998115e-17 with argmin = [0.9999999926662393, 0.9999999853324786] in 24 iterations
```

Note that this has fewer iterations.

As no derivative was given, it used **finite differences** to approximate the gradient of $f(x)$.

However, since most of the algorithms require derivatives, you will often want to use auto differentiation or pass analytical gradients if possible.

In [18]: $f(x) = (1.0 - x[1])^2 + 100.0 * (x[2] - x[1]^2)^2$

```
x_iv = [0.0, 0.0]
results = optimize(f, x_iv, LBFGS(), autodiff=:forward) # i.e. use ForwardDiff.jl
println("minimum = \$(results.minimum) with argmin = \$(results.minimizer)\n
minimum = 5.191703158437428e-27 with argmin = [0.999999999999928, 0.9999999999998559] in 24 iterations
```

Note that we did not need to use ForwardDiff.jl directly, as long as our $f(x)$ function was written to be generic (see the generic programming lecture).

Alternatively, with an analytical gradient

In [19]: $f(x) = (1.0 - x[1])^2 + 100.0 * (x[2] - x[1]^2)^2$

```
x_iv = [0.0, 0.0]
function g!(G, x)
end
results = optimize(f, g!, x_iv, LBFGS()) # or ConjugateGradient()
println("minimum = \$(results.minimum) with argmin = \$(results.minimizer)\n
minimum = 5.191703158437428e-27 with argmin = [0.999999999999928, 0.9999999999998559] in 24 iterations
```

Note that we did not need to use ForwardDiff.jl directly, as long as our $f(x)$ function was written to be generic (see the generic programming lecture).
For derivative-free methods, you can change the algorithm – and have no need to provide a gradient.

In [20]: \[ f(x) = (1.0 - x[1])^2 + 100.0 \times (x[2] - x[1]^2)^2 \]
\[
x_{\text{iv}} = [0.0, 0.0]
\]
\[
\text{results} = \text{optimize}(f, x_{\text{iv}}, \text{SimulatedAnnealing()} \ # \ or \ \text{ParticleSwarm()} \ or \ \text{NelderMead()})
\]

Out[20]:

* Status: failure (reached maximum number of iterations) (line search failed)

* Candidate solution
  
  Minimizer: [9.15e-01, 8.55e-01]
  Minimum: 3.770761e-02

* Found with
  
  Algorithm: Simulated Annealing
  Initial Point: [0.00e+00, 0.00e+00]

* Convergence measures
  
  \[ |x - x'| = \text{NaN} \quad 0.0e+00 \]
  \[ |x - x'|/|x'| = \text{NaN} \quad 0.0e+00 \]
  \[ |f(x) - f(x')| = \text{NaN} \quad 0.0e+00 \]
  \[ |f(x) - f(x')|/|f(x')| = \text{NaN} \quad 0.0e+00 \]
  \[ |g(x)| = \text{NaN} \quad 1.0e-08 \]

* Work counters
  
  Seconds run: 0 (vs limit Inf)
  Iterations: 1000
  \[ f(x) \text{ calls: 1001} \]

However, you will note that this did not converge, as stochastic methods typically require many more iterations as a tradeoff for their global-convergence properties.

See the maximum likelihood example and the accompanying Jupyter notebook.

**10.4.2 JuMP.jl**

The **JuMP.jl** package is an ambitious implementation of a modelling language for optimization problems in Julia. In that sense, it is more like an AMPL (or Pyomo) built on top of the Julia language with macros, and able to use a variety of different commercial and open source solvers.

If you have a linear, quadratic, conic, mixed-integer linear, etc. problem then this will likely be the ideal “meta-package” for calling various solvers.

For nonlinear problems, the modelling language may make things difficult for complicated functions (as it is not designed to be used as a general-purpose nonlinear optimizer).

See the quick start guide for more details on all of the options.

The following is an example of calling a linear objective with a nonlinear constraint (provided by an external function).

Here **Ipopt** stands for **Interior Point OPTimizer**, a nonlinear solver in Julia.
In [21]: using JuMP, Ipopt
    # solve
    # max( x[1] + x[2] )
    # st  sqrt(x[1]^2 + x[2]^2) <= 1

    function squareroot(x) # pretending we don't know sqrt()
        z = x # Initial starting point for Newton's method
        while abs(z*z - x) > 1e-13
            z = z - (z*z-x)/(2z)
        end
        return z
    end

    m = Model(with_optimizer(Ipopt.Optimizer))
    # need to register user defined functions for AD
    JuMP.register(m, :squareroot, 1, squareroot, autodiff=true)

    @variable(m, x[1:2], start=0.5) # start is the initial condition
    @objective(m, Max, sum(x))
    @NLconstraint(m, squareroot(x[1]^2 + x[2]^2) <= 1)
    @show JuMP.optimize!(m)

    Warning: `with_optimizer` is deprecated. Adapt the following example to your code:
    | `with_optimizer(Ipopt.Optimizer)` becomes `Ipopt.Optimizer`.
    | caller = top-level scope at In[21]:13
    | @ Core In[21]:13

This program contains Ipopt, a library for large-scale nonlinear optimization. 
Ipopt is released as open source code under the Eclipse Public License (EPL). 
For more information visit http://projects.coin-or.org/Ipopt

This is Ipopt version 3.12.10, running with linear solver mumps.
NOTE: Other linear solvers might be more efficient (see Ipopt documentation).

Number of nonzeros in equality constraint Jacobian...: 0
Number of nonzeros in inequality constraint Jacobian.: 2
Number of nonzeros in Lagrangian Hessian...: 3

Total number of variables...: 2
variables with only lower bounds: 0
variables with lower and upper bounds: 0
variables with only upper bounds: 0
Total number of equality constraints...: 0
Total number of inequality constraints...: 1
inequality constraints with only lower bounds: 0
inequality constraints with lower and upper bounds: 0
inequality constraints with only upper bounds: 1

iter  objective  inf_pr  inf_du  lg(mu) ||d||  lg(rg) alpha_du alpha_pr ls
0  -1.0000000e+00  0.00e+00  2.07e-01  -1.0  0.00e+00  -  0.00e+00  0.00e+00  0
1  -1.4100714e+00  0.00e+00  5.48e-02  -1.7  3.94e-01  -  1.00e+00  7.36e-01  1
2  -1.4140632e+00  0.00e+00  1.50e-08  -2.5  9.29e-04  -  1.00e+00  1.00e+00  1
3  -1.4142117e+00  0.00e+00  1.84e-11  -5.7  1.05e-03  -  1.00e+00  1.00e+00  1
4  -1.4142136e+00  0.00e+00  8.23e-09  -8.6  1.30e-06  -  1.00e+00  1.00e+00  1
5  -1.4142136e+00  0.00e+00  8.23e-09  -8.6  1.30e-06  -  1.00e+00  1.00e+00  1

iter  objective  inf_pr  inf_du  lg(mu) ||d||  lg(rg) alpha_du alpha_pr ls
0  -1.0000000e+00  0.00e+00  2.07e-01  -1.0  0.00e+00  -  0.00e+00  0.00e+00  0
1  -1.4100714e+00  0.00e+00  5.48e-02  -1.7  3.94e-01  -  1.00e+00  7.36e-01  1
2  -1.4140632e+00  0.00e+00  1.50e-08  -2.5  9.29e-04  -  1.00e+00  1.00e+00  1
3  -1.4142117e+00  0.00e+00  1.84e-11  -5.7  1.05e-03  -  1.00e+00  1.00e+00  1
4  -1.4142136e+00  0.00e+00  8.23e-09  -8.6  1.30e-06  -  1.00e+00  1.00e+00  1
5  -1.4142136e+00  0.00e+00  8.23e-09  -8.6  1.30e-06  -  1.00e+00  1.00e+00  1
Number of Iterations...: 5

(scaled) (unscaled)
Objective...: -1.4142135740093271e+00 -1.4142135740093271e+00
Dual infeasibility...: 8.2280586788385790e-09 8.2280586788385790e-09
Constraint violation...: 0.0000000000000000e+00 0.0000000000000000e+00
Complementarity...: 2.5059035815063646e-09 2.5059035815063646e-09
Overall NLP error...: 8.2280586788385790e-09 8.2280586788385790e-09

Number of objective function evaluations = 6
Number of objective gradient evaluations = 6
Number of equality constraint evaluations = 0
Number of inequality constraint evaluations = 6
Number of equality constraint Jacobian evaluations = 0
Number of inequality constraint Jacobian evaluations = 6
Number of Lagrangian Hessian evaluations = 5
Total CPU secs in IPOPT (w/o function evaluations) = 1.887
Total CPU secs in NLP function evaluations = 1.653
EXIT: Optimal Solution Found.
JuMP.optimize!(m) = nothing

And this is an example of a quadratic objective

In [22]: # solve
    # min (1-x)^2 + 100(y-x^2)^2
    # st x + y >= 10

    using JuMP, Ipopt
    m = Model(with_optimizer(Ipopt.Optimizer)) # settings for the solver
    @variable(m, x, start = 0.0)
    @variable(m, y, start = 0.0)

    @NLobjective(m, Min, (1-x)^2 + 100(y-x^2)^2)
    JuMP.optimize!(m)
    println("x = ", value(x), " y = ", value(y))

    # adding a (linear) constraint
    @constraint(m, x + y == 10)
    JuMP.optimize!(m)
    println("x = ", value(x), " y = ", value(y))

    └ Warning: `with_optimizer` is deprecated. Adapt the following example to update your code:
        `with_optimizer(Ipopt.Optimizer)` becomes `Ipopt.Optimizer`.
        caller = top-level scope at In[22]:6
    @Core In[22]:6

This is Ipopt version 3.12.10, running with linear solver mumps.
NOTE: Other linear solvers might be more efficient (see Ipopt documentation).

Number of nonzeros in equality constraint Jacobian...: 0
Number of nonzeros in inequality constraint Jacobian...: 0
10.4. OPTIMIZATION

Number of nonzeros in Lagrangian Hessian:. 3

Total number of variables:. 2
  variables with only lower bounds: 0
  variables with lower and upper bounds: 0
  variables with only upper bounds: 0

Total number of equality constraints:. 0

Total number of inequality constraints:. 0
  inequality constraints with only lower bounds: 0
  inequality constraints with lower and upper bounds: 0
  inequality constraints with only upper bounds: 0

iter  objective  inf_pr  inf_du  lg(mu)  ||d||  lg(rg)  alpha_du  alpha_pr  ls
0  1.0000000e+00  0.00e+00  2.00e+00  -1.0  0.00e+00  -0.00e+00 0
1  9.5312500e-01  0.00e+00  1.25e+01  -1.0  1.00e+00  -1.00e+00 2.50e-01 3
2  4.8320569e-01  0.00e+00  1.01e+00  -1.0  9.32e-02  -1.00e+00 1.00e+00 1
3  4.5708829e-01  0.00e+00  9.53e+00  -1.0  4.29e-01  -1.00e+00 5.00e-01 2
4  1.899205e-01  0.00e+00  4.15e-01  -1.0  9.51e-02  -1.00e+00 1.00e+00 1
5  1.3918726e-01  0.00e+00  6.51e+00  -1.7  3.49e-01  -1.00e+00 5.00e-01 2
6  5.4940999e-02  0.00e+00  4.51e-01  -1.7  9.29e-02  -1.00e+00 1.00e+00 1
7  2.944630e-02  0.00e+00  2.27e+00  -1.7  2.49e-01  -1.00e+00 5.00e-01 2
8  9.8586451e-03  0.00e+00  1.15e+00  -1.7  1.00e-01  -1.00e+00 1.00e+00 1
9  2.3237475e-03  0.00e+00  1.00e+00  -1.7  1.00e-01  -1.00e+00 1.00e+00 1
iter  objective  inf_pr  inf_du  lg(mu)  ||d||  lg(rg)  alpha_du  alpha_pr  ls
10  2.3797236e-04  0.00e+00  2.19e-01  -1.7  5.09e-02  -1.00e+00 1.00e+00 1
11  4.9267371e-06  0.00e+00  5.95e-02  -1.7  2.53e-02  -1.00e+00 1.00e+00 1
12  2.8189505e-09  0.00e+00  8.31e-04  -2.5  3.20e-03  -1.00e+00 1.00e+00 1
13  1.0095040e-15  0.00e+00  8.68e-07  -5.7  9.78e-05  -1.00e+00 1.00e+00 1
14  1.3288608e-28  0.00e+00  2.02e-13  -8.6  4.65e-08  -1.00e+00 1.00e+00 1

Number of Iterations:. 14

(scaled) (unscaled)
Objective:.  1.3288608467480825e-28  1.3288608467480825e-28
Dual infeasibility:.  2.0183854587685121e-13  2.0183854587685121e-13
Constraint violation:.  0.0000000000000000e+00  0.0000000000000000e+00
Complementarity:.  0.0000000000000000e+00  0.0000000000000000e+00
Overall NLP error:.  2.0183854587685121e-13  2.0183854587685121e-13

Number of objective function evaluations = 36
Number of objective gradient evaluations = 15
Number of equality constraint evaluations = 0
Number of inequality constraint evaluations = 0
Number of equality constraint Jacobian evaluations = 0
Number of inequality constraint Jacobian evaluations = 0
Number of Lagrangian Hessian evaluations = 14
Total CPU secs in IPOPT (w/o function evaluations) = 0.085
Total CPU secs in NLP function evaluations = 0.013

EXIT: Optimal Solution Found.
x = 0.9999999999999999 y = 0.9999999999999999
This is Ipopt version 3.12.10, running with linear solver mumps.
NOTE: Other linear solvers might be more efficient (see Ipopt documentation).

Number of nonzeros in equality constraint Jacobian:. 2
Number of nonzeros in inequality constraint Jacobian:. 0
Number of nonzeros in Lagrangian Hessian:. 3

Total number of variables:. 2
  variables with only lower bounds: 0
  variables with lower and upper bounds: 0
  variables with only upper bounds: 0
Total number of equality constraints: 1
Total number of inequality constraints: 0
inequality constraints with only lower bounds: 0
inequality constraints with lower and upper bounds: 0
inequality constraints with only upper bounds: 0

iter objective inf_pr inf_du lg(mu) ||d|| lg(rg) alpha_du alpha_pr ls
0 1.0000000e+00 1.00e+01 1.00e+00 -1.0 0.00e+00 0.00e+00 0 0
1 9.6315968e+05 0.00e+00 3.89e+05 -1.0 9.91e+00 -1.00e+00 1.00e+00 1
2 1.6901461e+05 0.00e+00 1.16e+05 -1.0 3.24e+00 -1.00e+00 1.00e+00 1
3 2.5433173e+04 1.78e-15 3.18e+04 -1.0 2.05e+00 -1.00e+00 1.00e+00 1
4 5.1380324e+02 0.00e+00 1.35e+03 -1.0 5.62e-01 -1.00e+00 1.00e+00 1
5 3.3745556e+00 0.00e+00 8.45e+01 -1.0 1.50e-01 -1.00e+00 1.00e+00 1
6 2.8946796e+00 0.00e+00 4.22e-01 -1.0 1.07e-02 -1.00e+00 1.00e+00 1
7 2.8946796e+00 0.00e+00 1.08e-05 -1.7 5.42e-05 -1.00e+00 1.00e+00 1
8 2.8946796e+00 0.00e+00 1.08e-05 -1.7 5.42e-05 -1.00e+00 1.00e+00 1
9 2.8946796e+00 0.00e+00 1.08e-05 -1.7 5.42e-05 -1.00e+00 1.00e+00 1

Number of Iterations: 9

Objective (scaled) (unscaled)
2.8946075504894599e+00 2.8946075504894599e+00
Dual infeasibility (scaled) (unscaled)
5.9130478291535837e-13 5.9130478291535837e-13
Constraint violation (scaled) (unscaled)
0.0000000000000000e+00 0.0000000000000000e+00
Complementarity (scaled) (unscaled)
0.0000000000000000e+00 0.0000000000000000e+00
Overall NLP error (scaled) (unscaled)
5.9130478291535837e-13 5.9130478291535837e-13

Number of objective function evaluations = 10
Number of objective gradient evaluations = 10
Number of equality constraint evaluations = 10
Number of inequality constraint evaluations = 0
Number of equality constraint Jacobian evaluations = 1
Number of inequality constraint Jacobian evaluations = 0
Number of Lagrangian Hessian evaluations = 9
Total CPU secs in IPOPT (w/o function evaluations) = 0.003
Total CPU secs in NLP function evaluations = 0.000

EXIT: Optimal Solution Found.

x = 2.701147124098218
y = 7.2988528759017814

10.4.3 BlackBoxOptim.jl

Another package for doing global optimization without derivatives is BlackBoxOptim.jl.

To see an example from the documentation

In [23]: using BlackBoxOptim

    function rosenbrock2d(x)
        return (1.0 - x[1])^2 + 100.0 * (x[2] - x[1]^2)^2
    end

    results = bboptimize(rosenbrock2d; SearchRange = (-5.0, 5.0), NumDimensions = 2);

    Starting optimization with optimizer
    DiffEvoOpt{FitPopulation{Float64},RadiusLimitedSelector,BlackBoxOptim.AdaptiveDiffEvoRandBin{3},RandomBound{ContinuousRectSearchSpace}}
10.5. SYSTEMS OF EQUATIONS AND LEAST SQUARES

0.00 secs, 0 evals, 0 steps

Optimization stopped after 10001 steps and 0.09 seconds
Termination reason: Max number of steps (10000) reached
Steps per second = 115848.82
Function evals per second = 117157.78
Improvements/step = 0.20700
Total function evaluations = 10114

Best candidate found: [1.0, 1.0]
Fitness: 0.000000000

An example for parallel execution of the objective is provided.

10.5 Systems of Equations and Least Squares

10.5.1 Roots.jl

A root of a real function $f$ on $[a, b]$ is an $x \in [a, b]$ such that $f(x) = 0$.

For example, if we plot the function

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

with $x \in [0, 1]$ we get

![Graph of the function](image)

The unique root is approximately 0.408.

The Roots.jl package offers `fzero()` to find roots.
In [24]: using Roots
   f(x) = sin(4 * (x - 1/4)) + x + x^20 - 1
   fzero(f, 0, 1)

Out[24]: 0.40829350427936706

10.5.2 NLsolve.jl

The NLsolve.jl package provides functions to solve for multivariate systems of equations and fixed points.

From the documentation, to solve for a system of equations without providing a Jacobian

In [25]: using NLsolve
   f(x) = [(x[1]+3)*(x[2]^3-7)+18
          sin(x[2]*exp(x[1])-1)] # returns an array
   results = nlsolve(f, [ 0.1; 1.2])

Out[25]: Results of Nonlinear Solver Algorithm
   * Algorithm: Trust-region with dogleg and autoscaling
   * Starting Point: [0.1, 1.2]
   * Zero: [-7.775548712324193e-17, 0.9999999999999999]
   * Inf-norm of residuals: 0.000000
   * Iterations: 4
   * Convergence: true
     * |x - x'| < 0.0e+00: false
     * |f(x)| < 1.0e-08: true
   * Function Calls (f): 5
   * Jacobian Calls (df/dx): 5

In the above case, the algorithm used finite differences to calculate the Jacobian.

Alternatively, if f(x) is written generically, you can use auto-differentiation with a single setting.

In [26]: results = nlsolve(f, [ 0.1; 1.2], autodiff=:forward)

   println("converged=$\{NLsolve.converged(results)\} at root=$\{results.zero\]"
   "$\{results.iterations\} iterations and $\{results.f_calls\} function calls")

   converged=true at root=[-3.487552479724522e-16, 1.0000000000000002] in 4
      iterations
   and 5 function calls

Providing a function which operates inplace (i.e., modifies an argument) may help performance for large systems of equations (and hurt it for small ones).

In [27]: function f!(F, x) # modifies the first argument
F[2] = sin(x[2]*exp(x[1])-1)

end

results = nlsolve(f!, [ 0.1; 1.2], autodiff=:forward)

println("converged=$NLsolve.converged(results)) at root=$(results.zero)"

println("$NLsolve.iterations iterations and $(results.f_calls) function calls")

converged=true at root=[-3.487552479724522e-16, 1.0000000000000002e-16] in 4 iterations
and 5 function calls

10.6 LeastSquaresOptim.jl

Many optimization problems can be solved using linear or nonlinear least squares.

Let \( x \in \mathbb{R}^N \) and \( F(x) : \mathbb{R}^N \rightarrow \mathbb{R}^M \) with \( M \geq N \), then the nonlinear least squares problem is

\[
\min_x F(x)^T F(x)
\]

While \( F(x)^T F(x) \rightarrow \mathbb{R} \), and hence this problem could technically use any nonlinear optimizer, it is useful to exploit the structure of the problem.

In particular, the Jacobian of \( F(x) \), can be used to approximate the Hessian of the objective.

As with most nonlinear optimization problems, the benefits will typically become evident only when analytical or automatic differentiation is possible.

If \( M = N \) and we know a root \( F(x^*) = 0 \) to the system of equations exists, then NLS is the defacto method for solving large systems of equations.

An implementation of NLS is given in LeastSquaresOptim.jl.

From the documentation

In [28]: using LeastSquaresOptim
   function rosenbrock(x)
      [1 - x[1], 100 * (x[2]-x[1]^2)]
   end
   LeastSquaresOptim.optimize(rosenbrock, zeros(2), Dogleg())

Out[28]:
   Results of Optimization Algorithm
   * Algorithm: Dogleg
   * Minimizer: [1.0,1.0]
   * Sum of squares at Minimum: 0.000000
   * Iterations: 51
   * Convergence: true
   * |x - x'| < 1.0e-08: false
   * |f(x) - f(x')| / |f(x)| < 1.0e-08: true
   * |g(x)| < 1.0e-08: false
   * Function Calls: 52
   * Gradient Calls: 36
   * Multiplication Calls: 159
Note: Because there is a name clash between Optim.jl and this package, to use both we need to qualify the use of the optimize function (i.e. LeastSquaresOptim.optimize).

Here, by default it will use AD with ForwardDiff.jl to calculate the Jacobian, but you could also provide your own calculation of the Jacobian (analytical or using finite differences) and/or calculate the function inplace.

```
function rosenbrock_f!(out, x)
    out[1] = 1 - x[1]
end

LeastSquaresOptim.optimize!(LeastSquaresProblem(x = zeros(2),
                          f! = rosenbrock_f!, output_length = 2))
```

```
# if you want to use gradient
function rosenbrock_g!(J, x)
    J[1, 1] = -1
    J[1, 2] = 0
    J[2, 1] = -200 * x[1]
    J[2, 2] = 100
end

LeastSquaresOptim.optimize!(LeastSquaresProblem(x = zeros(2),
                          f! = rosenbrock_f!, g! = rosenbrock_g!,
                          output_length = 2))
```

Out[29]: Results of Optimization Algorithm

- Algorithm: Dogleg
- Minimizer: [1.0, 1.0]
- Sum of squares at Minimum: 0.000000
- Iterations: 51
- Convergence: true
- |x - x'| < 1.0e-08: false
- |f(x) - f(x')| / |f(x)| < 1.0e-08: true
- |g(x)| < 1.0e-08: false
- Function Calls: 52
- Gradient Calls: 36
- Multiplication Calls: 159

### 10.7 Additional Notes

Watch this video from one of Julia’s creators on automatic differentiation.

### 10.8 Exercises

#### 10.8.1 Exercise 1

Doing a simple implementation of forward-mode auto-differentiation is very easy in Julia since it is generic. In this exercise, you will fill in a few of the operations required for a simple AD implementation.

First, we need to provide a type to hold the dual.
In [30]: struct DualNumber{T} <: Real
    val::T
    ϵ::T
end

Here we have made it a subtype of Real so that it can pass through functions expecting Reals.

We can add on a variety of chain rule definitions by importing in the appropriate functions
and adding DualNumber versions. For example

In [31]: import Base: +, *, -, ^, exp
+(x::DualNumber, y::DualNumber) = DualNumber(x.val + y.val, x.ϵ + y.ϵ) #
   dual addition
+(x::DualNumber, a::Number) = DualNumber(x.val + a, x.ϵ) # i.e. scalar
   addition, not
dual
+(a::Number, x::DualNumber) = DualNumber(x.val + a, x.ϵ) # i.e. scalar
   addition, not
dual

Out[31]: + (generic function with 265 methods)

With that, we can seed a dual number and find simple derivatives,

In [32]: f(x, y) = 3.0 + x + y
x = DualNumber(2.0, 1.0) # x -> 2.0 + 1.0\epsilon
y = DualNumber(3.0, 0.0) # i.e. y = 3.0, no derivative

# seeded calculates both teh function and the d/dx gradient!
f(x,y)

Out[32]: DualNumber{Float64}(8.0, 1.0)

For this assignment:

1. Add in AD rules for the other operations: *, -, ^, exp.
2. Come up with some examples of univariate and multivariate functions combining those
   operations and use your AD implementation to find the derivatives.
CHAPTER 10. SOLVERS, OPTIMIZERS, AND AUTOMATIC DIFFERENTIATION
Chapter 11

Julia Tools and Editors

11.1 Contents

- Preliminary Setup 11.2
- The REPL 11.3
- Atom 11.4
- Package Environments 11.5

Co-authored with Arnav Sood

While Jupyter notebooks are a great way to get started with the language, eventually you will want to use more powerful tools.

We’ll discuss a few of them here, such as

- Text editors like Atom, which come with rich Julia support for debugging, documentation, git integration, plotting and inspecting data, and code execution.
- The Julia REPL, which has specialized modes for package management, shell commands, and help.

Note that we assume you’ve already completed the getting started and interacting with Julia lectures.

11.2 Preliminary Setup

Follow the instructions for setting up Julia on your local computer.

11.3 The REPL

Previously, we discussed basic use of the Julia REPL (“Read-Evaluate-Print Loop”).

Here, we’ll consider some more advanced features.

11.3.1 Shell Mode

Hitting ; brings you into shell mode, which lets you run bash commands (PowerShell on Windows)
In [1]: ; pwd

/home/ubuntu/repos/lecture-source-jl/_build/jupyterpdf/executed/more_julia

You can also use Julia variables from shell mode

In [2]: x = 2

Out[2]: 2

In [3]: ; echo $x

 2

### 11.3.2 Package Mode

Hitting ] brings you into package mode.

- ] add **Expectations** will add a package (here, **Expectations.jl**).
- Likewise, ] rm **Expectations** will remove that package.
- ] st will show you a snapshot of what you have installed.
- ] up will (intelligently) upgrade versions of your packages.
- ] precompile will precompile everything possible.
- ] build will execute build scripts for all packages.
- Running ] preview before a command (i.e., ] preview up) will display the changes without executing.

You can get a full list of package mode commands by running

In [4]: ] ?

Welcome to the Pkg REPL-mode. To return to the julia> prompt, either press backspace when the input line is empty or press Ctrl+C.

**Synopsis**

pkg> cmd [opts] [args]

Multiple commands can be given on the same line by interleaving a ; between the commands.

**Commands**

activate: set the primary environment the package manager manipulates
add: add packages to project
build: run the build script for packages
develop: clone the full package repo locally for development
free: undoes a pin, develop, or stops tracking a repo

gc: garbage collect packages not used for a significant time

generate: generate files for a new project

help: show this message

instantiate: downloads all the dependencies for the project

pin: pins the version of packages

precompile: precompile all the project dependencies

redo: redo the latest change to the active project

remove: remove packages from project or manifest

resolve: resolves to update the manifest from changes in dependencies of developed packages

status: summarize contents of and changes to environment

test: run tests for packages

undo: undo the latest change to the active project

update: update packages in manifest

registry add: add package registries

registry remove: remove package registries

registry status: information about installed registries

registry update: update package registries

On some operating systems (such as OSX) REPL pasting may not work for package mode, and you will need to access it in the standard way (i.e., hit ] first and then run your commands).

11.3.3 Help Mode

Hitting ? will bring you into help mode.

The key use case is to find docstrings for functions and macros, e.g.

? print

Note that objects must be loaded for Julia to return their documentation, e.g.

? @test

will fail, but

using Test
? @test

will succeed.

11.4 Atom

As discussed previously, eventually you will want to use a fully fledged text editor. The most feature-rich one for Julia development is Atom, with the Juno package.

There are several reasons to use a text editor like Atom, including

- Git integration (more on this in the next lecture).
- Painless inspection of variables and data.
- Easily run code blocks, and drop in custom snippets of code.
- Integration with Julia documentation and plots.

11.4.1 Installation and Configuration

Installing Atom

1. Download and Install Atom from the Atom website.

2. (Optional, but recommended): Change default Atom settings
   - Use Ctrl-, to get the Settings pane
   - Choose the Packages tab
   - Type line-ending-selector into the Filter and then click “Settings” for that package
     - Change the default line ending to LF (only necessary on Windows)
   - Choose the Editor tab
     - Turn on Soft Wrap
     - Set the Tab Length default to 4

Installing Juno

1. Use Ctrl-, to get the Settings pane.

2. Go to the Install tab.

3. Type uber-juno into the search box and then click Install on the package that appears.

4. Wait while Juno installs dependencies.

5. When it asks you whether or not to use the standard layout, click yes.

At that point, you should see a built-in REPL at the bottom of the screen and be able to start using Julia and Atom.
Troubleshooting

Sometimes, Juno will fail to find the Julia executable (say, if it’s installed somewhere non-standard, or you have multiple).

To do this 1. Ctrl-, to get Settings pane, and select the Packages tab. 2. Type in julia-client and choose Settings. 3. Find the Julia Path, and fill it in with the location of the Julia binary.

- To find the binary, you could run `Sys.BINDIR` in the REPL, then add in an additional `/julia` to the end of the screen.

  e.g. `C:\Users\YOURUSERNAME\AppData\Local\Julia-1.0.1\bin\julia.exe` on Windows as `/Applications/Julia-1.0.app/Contents/Resources/julia/bin/julia` on OSX.

See the setup instructions for Juno if you have further issues.

If you upgrade Atom and it breaks Juno, run the following in a terminal.

```
apm uninstall ink julia-client
apm install ink julia-client
```

If you aren’t able to install `apm` in your PATH, you can do the above by running the following in PowerShell:

```
cd $ENV:LOCALAPPDATA/atom/bin
```

Then navigating to a folder like `C:\Users\USERNAME\AppData\Local\atom\bin` (which will contain the `apm` tool), and running:

```
./apm uninstall ink julia-client
./apm install ink julia-client
```

Upgrading Julia

To get a new release working with Jupyter, run (in the new version’s REPL)

```
] add IJulia
] build IJulia
```

This will install (and build) the IJulia kernel.

To get it working with Atom, open the command palette and type “Julia Client: Settings.”

Then, in the box labelled “Julia Path,” enter the path to your Julia executable.

You can find the folder by running `Sys.BINDIR` in a new REPL, and then add the `/julia` at the end to give the exact path.

For example:
11.4.2 Standard Layout

If you follow the instructions, you should see something like this when you open a new file. If you don’t, simply go to the command palette and type “Julia standard layout”

The bottom pane is a standard REPL, which supports the different modes above. The “workspace” pane is a snapshot of currently-defined objects. For example, if we define an object in the REPL

\[
\text{In [5]: } x = 2
\]

\[
\text{Out[5]: } 2
\]

Our workspace should read

The `ans` variable simply captures the result of the last computation.
11.4. ATOM

The **Documentation** pane simply lets us query Julia documentation

```
using Plots
gr(fmt = :png);
data = rand(10, 10)
h = heatmap(data)
```

**Note:** The plots feature is not perfectly reliable across all plotting backends, see the Basic Usage page.
11.4.3 Other Features

- **Shift + Enter** will evaluate a highlighted selection or line (as above).
- The run symbol in the left sidebar (or `Ctrl+Shift+Enter`) will run the whole file.

See basic usage for an exploration of features, and the FAQ for more advanced steps.

11.5 Package Environments

Julia’s package manager lets you set up Python-style “virtualenvs,” or subsets of packages that draw from an underlying pool of assets on the machine.

This way, you can work with (and specify) the dependencies (i.e., required packages) for one project without worrying about impacts on other projects.

- An environment is a set of packages specified by a `Project.toml` (and optionally, a `Manifest.toml`).
- A registry is a git repository corresponding to a list of (typically) registered packages, from which Julia can pull (for more on git repositories, see version control).
- A depot is a directory, like `~/.julia`, which contains assets (compile caches, registries, package source directories, etc.).

Essentially, an environment is a dependency tree for a project, or a “frame of mind” for Julia’s package manager.

- We can see the default (`v1.1`) environment as such

```
In [6]: ] st
```

```
 Status `~/repos/lecture-source-jl/_build/jupyterpdf/executed/more_julia/Project.toml`
 [2169fc97] AlgebraicMultigrid v0.2.2
 [28f2cc6d] ApproxFun v0.11.13
 [7d9fca2a] Arpack v0.4.0
 [aae01518] BandedMatrices v0.15.7
 [6e4bb0f9] BenchmarkTools v0.5.0
 [a134a8b2] BlackBoxOptim v0.5.0
 [ffab5731] BlockBandedMatrices v0.8.4
 [324d7699] CategoricalArrays v0.8.0
 [34da2185] Compat v2.2.0
 [a93c6f00] DataFrames v0.21.0
 [1313f7d8] DataFramesMeta v0.5.1
 [39dd38d3] Dierckx v0.4.1
 [9fddde737] DiffEqOperators v4.10.0
 [31c24e18] Distributions v0.23.2
 [2fe49d83] Expectations v1.1.1
 [a1e7a1ef] Expokit v0.2.0
 [d4dd017d3] ExponentialUtilities v1.6.0
 [442a2c76] FastGaussQuadrature v0.4.2
 [1a297f60] FillArrays v0.8.9
 [95d5cd8c9] FixedEffectModels v0.10.7
 [c8885935] FixedEffects v0.7.3
 [587475ba] Flux v0.10.4
 [7f6369f11] ForwardDiff v0.10.10
 [38e38edf] GLM v1.3.0
 [28bb8d3ca] GR v0.49.1
 [4071384e] IncompleteLU v0.1.1
 [43edad99] InstantiateFromURL v0.5.0
```
We can also create and activate a new environment

In [7]: generate ExampleEnvironment

Generating project ExampleEnvironment:
ExampleEnvironment/Project.toml
ExampleEnvironment/src/ExampleEnvironment.jl

In [8]: ; cd ExampleEnvironment
To activate the directory, simply

```
In [9]: ] activate .
```

Activating environment at `~/repos/lecture-source-jl/_build/jupyterpdf/executed/more_julia/ExampleEnvironment/Project.toml`

where “.” stands in for the “present working directory”.

Let’s make some changes to this

```
In [10]: ] add Expectations Parameters
```

Updating registry at `~/.julia/registries/General`

Updating git-repo
`https://github.com/JuliaRegistries/General.git`

Resolving package versions...

```
Updating `~/repos/lecture-source-jl/_build/jupyterpdf/executed/more_julia/ExampleEnvironment/Project.toml`
[2fe49d83] + Expectations v1.6.0
[d96e819e] + Parameters v0.12.1

Updating `~/repos/lecture-source-jl/_build/jupyterpdf/executed/more_julia/ExampleEnvironment/Manifest.toml`
[56f22d72] + Artifacts v1.3.0
[34da2185] + Compat v3.23.0
[e66e007b] + CompilerSupportLibraries_jll v0.3.4+0
[9a962f9c] + DataAPI v1.4.0
[864edeb3b] + DataStructures v0.18.8
[31c24ae10] + Distributions v0.23.12
[2fe49d83] + Expectations v1.6.0
[442a2c76] + FastGaussQuadrature v0.4.4
[1a297fd60] + FillArrays v0.9.7
[692b3bcd] + JLWrappers v1.1.3
[e1d29d7a] + Missings v0.4.4
[efe28fd5] + OpenSpecFun_jll v0.5.3+4
[ bac558e1] + OrderedCollections v1.3.2
[90014a1f] + PDMats v0.10.1
[d96e819e] + Parameters v0.12.1
[1fd47b50] + QuadGK v2.4.1
[79098fc4] + Rmath v0.6.1
[f50d1b31] + Rmath_jll v0.2.2+1
[a2af1166] + SortingAlgorithms v0.3.1
[276daf66] + SpecialFunctions v0.10.3
[90137ffa] + StaticArrays v0.12.5
```
Note the lack of commas

- To see the changes, simply open the `ExampleEnvironment` directory in an editor like Atom.

The Project TOML should look something like this

```
name = "ExampleEnvironment"
uuid = "14d3e79e-e2e5-11e8-28b9-19823016c34c"
authors = ["QuantEcon User <quanteconuser@gmail.com>"]
version = "0.1.0"

[deps]
Expectations = "2fe49d83-0758-5602-8f54-1f90ad0d522b"
Parameters = "d96e819e-fc66-5662-9728-84c9c7592b0a"
```

We can also

```
In [11]: ] precompile
```

```
  Precompiling project...
  Info: Precompiling ExampleEnvironment [d548e192-c78c-496f-9c49-1ae7ad370b2e]
  @ Base loading.jl:1260
```

Note The TOML files are independent of the actual assets (which live in `~/.julia/packages`, `~/.julia/dev`, and `~/.julia/compiled`).
You can think of the TOML as specifying demands for resources, which are supplied by the
~/.julia user depot.

- To return to the default Julia environment, simply

In [12]: ] activate

    Activating environment at `~/repos/lecture-source-jl/_build/jupyterpdf/executed/more_julia/Project.toml`

without any arguments.

- Lastly, let’s clean up

In [13]: ; cd ..

/home/ubuntu/repos/lecture-source-jl/_build/jupyterpdf/executed/more_julia

In [14]: ; rm -rf ExampleEnvironment

11.5.1 InstantiateFromURL

With this knowledge, we can explain the operation of the setup block

In [15]: using InstantiateFromURL

    # optionally add arguments to force installation: instantiate = true, 
    ~precompile = true
    github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

What this github_project function does is activate (and if necessary, download, instantiate and precompile) a particular Julia environment.
Chapter 12

Git, GitHub, and Version Control

12.1 Contents

- Setup 12.2
- Basic Objects 12.3
- Individual Workflow 12.4
- Collaborative Work 12.5
- Collaboration via Pull Request 12.6
- Additional Resources and Troubleshooting 12.7
- Exercises 12.8

Co-authored with Arnav Sood

An essential part of modern software engineering is using version control.

We use version control because

- Not all iterations on a file are perfect, and you may want to revert changes.
- We want to be able to see who has changed what and how.
- We want a uniform version scheme to do this between people and machines.
- Concurrent editing on code is necessary for collaboration.
- Version control is an essential part of creating reproducible research.

In this lecture, we’ll discuss how to use Git and GitHub.

12.2 Setup

1. Make sure you create an account on GitHub.com.

2. If you are a student, be sure to use the GitHub Student Developer Pack. Otherwise, see if you qualify for a free Non-Profit/Academic Plan. These come with things like unlimited private repositories, testing support, etc.

3. Install git and the GitHub Desktop application.

4. Install git.

5. Install the GitHub Desktop application.
4. Optionally (but strongly recommended): On Windows, change the default line-ending by:

5. Opening a Windows/Powershell console, or the “Git Bash” installed in the previous step.

6. Running the following

```bash
git config --global core.eol lf
git config --global core.autocrlf false
```

### 12.2.1 Git vs. GitHub vs. GitHub Desktop

To understand the relationship

- Git is an infrastructure for versioning and merging files (it is not specific to GitHub and does not even require an online server).
- GitHub provides an online service to coordinate working with Git repositories, and adds some additional features for managing projects.
- GitHub Desktop is just one of many GUI-based clients to make Git and GitHub easier to use.

Later, you may find yourself using alternatives

- GitHub is the market leader for open source projects and Julia, but there are other options, e.g. GitLab and Bitbucket.
- Instead of the GitHub Desktop, you may directly use the Git command line, GitKraken, or use the Git functionality built into editors such as Atom or VS Code.

Since these lecture notes are intended to provide a minimal path to using the technologies, here we will conflate the workflow of these distinct products.

### 12.3 Basic Objects

#### 12.3.1 Repositories

The fundamental object in GitHub is a *repository* (or “repo”) – this is the master directory for a project.

One example of a repo is the QuantEcon `Expectations.jl` package.

On the machine, a repo is a normal directory, along with a subdirectory called `.git` which contains the history of changes.

#### 12.3.2 Commits

GitHub stores history as a sequence of changes to text, called *commits*.

Here is an example of a commit, which revises the style guide in a QuantEcon repo.

In particular, commits have the following features

- An ID (formally, an “SHA-1 hash”)
- Content (i.e., a before and after state)
12.4. INDIVIDUAL WORKFLOW

- Metadata (author, timestamp, commit message, etc.)

**Note:** It’s crucial to remember that what’s stored in a commit is only the actual changes you make to text.

This is a key reason why git can store long and complicated histories without consuming massive amounts of memory.

12.3.3 Common Files

In addition, each GitHub repository typically comes with a few standard text files

- A `.gitignore` file, which lists files/extensions/directories that GitHub shouldn’t try to track (e.g., LaTeX compilation byproducts).
- A `README.md` file, which is a Markdown file which GitHub puts on the repository website.
- A `LICENSE.txt` file, which describes the terms under which the repository’s contents are made available.

For an example of all three, see the Expectations.jl repo.

Of these, the `README.md` is the most important, as GitHub will display it as Markdown when accessing the repository online.

12.4 Individual Workflow

In this section, we’ll describe how to use GitHub to version your own projects.

Much of this will carry over to the collaborative section.

12.4.1 Creating a Repository

In general, we will always want to repos for new projects using the following dropdown

![Dropdown menu for creating a repository](image)

We can then configure repository options as such
In this case, we’re making a public repo `github.com/quantecon_user/example_repository`, which will come with a `README.md`, is licensed under the MIT License, and will ignore Julia compilation byproducts.

**Note** This workflow is for creating projects *de novo*; the process for turning existing directories into git repos is a bit more complicated.

In particular, in that case we recommend that you create a new repo via this method, then copy in and commit your files (see below), and then delete the old directory.

### 12.4.2 Cloning a Repository

The next step is to get this to our local machine

This dropdown gives us a few options

- “Open in Desktop” will call to the GitHub Desktop application that we’ve installed.
- “Download Zip” will download the directory *without the*.git subdirectory (avoid this option).
- The copy/paste button next to the link lets us use the command line, i.e. `git clone https://github.com/quanteconuser/example_repository.git`. 

...
12.4.3 Making and Managing Changes

Now that we have the repository, we can start working with it.

For example, let’s say that we’ve amended the README.md (using our editor of choice), and also added a new file economics.jl which we’re still working on.

Returning to GitHub Desktop, we should see something like

To select individual files for commit, we can use the check boxes to the left of each file.

Let’s say you select only the README to commit. Going to the history tab should show you our change

The Julia file is unchanged.

12.4.4 Pushing to the Server

As of now, this commit lives only on our local machine.

To upload it to the server, you can simply click the “Push Origin” button at the top the screen.
The small “1” to the right of the text indicates we have one commit to upload.

12.4.5 Reading and Reverting History

As mentioned, one of the key features of GitHub is the ability to scan through history. By clicking the “commits” tab on the repo front page, we see this page (as an example). Clicking an individual commit gives us the difference view, (e.g., example commit).

Sometimes, however, we want to not only inspect what happened before, but reverse the commit.

- If you haven’t made the commit yet, just right-click the file in the “changes” tab and hit “discard changes” to reset the file to the last known commit.
- If you have made the commit but haven’t pushed to the server yet, go to the “history” tab as above, right click the commit and click “revert this commit.” This will create the inverse commit, shown below.

12.4.6 Working across Machines

Generally, you want to work on the same project but across multiple machines (e.g., a home laptop and a lab workstation).

The key is to push changes from one machine, and then to pull changes from the other machine.

Pushing can be done as above.

To pull, simply click pull under the “repository” dropdown at the top of the screen.

12.5 Collaborative Work

12.5.1 Adding Collaborators

First, let’s add a collaborator to the quanteconuser/example_repository lecture we created earlier.
12.5. COLLABORATIVE WORK

We can do this by clicking “settings => collaborators,” as follows

12.5.2 Project Management

GitHub’s website also comes with project management tools to coordinate work between people.

The main one is an issue, which we can create from the issues tab.

You should see something like this

Let’s unpack the different components

- The **assignees** dropdown lets you select people tasked to work on the issue.
- The **labels** dropdown lets you tag the issue with labels visible from the issues page, such as “high priority” or “feature request”.
- It’s possible to tag other issues and collaborators (including in different repos) by linking to them in the comments – this is part of what’s called *GitHub-Flavored Markdown*.

For an example of an issue, see here.

You can see open issues at a glance from the general issues tab.
The checkboxes are common in GitHub to manage project tasks.

12.5.3 Reviewing Code

There are a few different ways to review people’s code in GitHub:

- Whenever people push to a project you’re working on, you’ll receive an email notification.
- You can also review individual line items or commits by opening commits in the difference view as above.

12.5.4 Merge Conflicts

Any project management tool needs to figure out how to reconcile conflicting changes between people.

In GitHub, this event is called a “merge conflict,” and occurs whenever people make conflicting changes to the same line of code.

Note that this means that two people touching the same file is OK, so long as the differences are compatible.

A common use case is when we try to push changes to the server, but someone else has pushed conflicting changes.

GitHub will give us the following window
12.6. COLLABORATION VIA PULL REQUEST

- The warning symbol next to the file indicates the existence of a merge conflict.
- The viewer tries to show us the discrepancy (I changed the word repository to repo, but someone else tried to change it to “repo” with quotes).

To fix the conflict, we can go into a text editor (such as Atom)

Let’s say we click the first “use me” (to indicate that my changes should win out), and then save the file.

Returning to GitHub Desktop gives us a pre-formed commit to accept

Clicking “commit to master” will let us push and pull from the server as normal.

12.6 Collaboration via Pull Request

One of the defining features of GitHub is that it is the dominant platform for open source code, which anyone can access and use.
However, while anyone can make a copy of the source code, not everyone has access to modify the particular version stored on GitHub.

A maintainer (i.e. someone with “write” access to directly modify a repository) might consider different contributions and “merge” the changes into the main repository if the changes meet their criteria.

A pull request (“PR”) allows any outsiders to suggest changes to open source repositories. A PR requests the project maintainer to merge (“pull”) changes you’ve worked on into their repository.

There are a few different workflows for creating and handling PRs, which we’ll walk through below.

**Note:** If the changes are for a Julia Package, you will need to follow a different workflow – described in the testing lecture.

### 12.6.1 Quick Fixes

GitHub’s website provides an online editor for quick and dirty changes, such as fixing typos in documentation.

To use it, open a file in GitHub and click the small pencil to the upper right

Here, we’re trying to add the QuantEcon link to the Julia project’s README file.

After making our changes, we can then describe and propose them for review by maintainers.

But what if we want to make more in-depth changes?

### 12.6.2 No-Access Case

A common problem is when we don’t have write access (i.e. we can’t directly modify) the repo in question.

In that case, click the “Fork” button that lives in the top-right of every repo’s main page
This will copy the repo into your own GitHub account.

For example, this repo is a fork of our original git setup.

Clone this fork to our desktop and work with it in exactly the same way as we would a repo we own (as the fork is in your account, you now have write access).

That is, click the “clone” button on our fork

You’ll see a new repo with the same name but different URL in your GitHub Desktop repo list, along with a special icon to indicate that it’s a fork

Commit some changes by selecting the files and writing a commit message
And push by using the dropdown
Below, for example, we’ve committed and pushed some changes to the fork that we want to upstream into the main repo.

We should make sure that these changes are on the server (which we can get to by going to the fork and clicking “commits”).

Next, go to the pull requests menu and click “New Pull Request.” You’ll see something like this:

This gives us a quick overview of the commits we want to merge in, as well as the overall differences. Hit create and then click through the following form.
CHAPTER 12. GIT, GITHUB, AND VERSION CONTROL

This opens a page like this on the main repo

![Example Pull Request]

The key pieces are

- A list of the commits we’re proposing.
- A list of reviewers, who can approve or modify our changes.
- Labels, Markdown space, assignees, and the ability to tag other git issues and PRs, just as with issues.

Here’s an example pull request.

To edit a PR, simply push changes to the fork you cloned to your desktop.

For example, let’s say we commit a new change to the README after we create the PR

![Commit to Readme]

After pushing to the server, the change is reflected on the PR page

![Commit Reflection]

That is, creating a pull request is not like bundling up your changes and delivering them, but rather like opening an ongoing connection between two repositories, that is only severed when the PR is closed or merged.
12.6.3 Write Access Case

As you become more familiar with GitHub, and work on larger projects, you will find yourself making PRs even when it isn’t strictly required.

If you are a maintainer of the repo (e.g. you created it or are a collaborator) then you don’t need to create a fork, but will rather work with a *git branch*.

Branches in git represent parallel development streams (i.e., sequences of commits) that the PR is trying to merge.

First, load the repo in GitHub Desktop and use the branch dropdown

Click “New Branch” and choose an instructive name (make sure there are no spaces or special characters).

This will “check out” a new branch with the same history as the old one (but new commits will be added only to this branch).

We can see the active branch in the top dropdown

For example, let’s say we add some stuff to the Julia code file and commit it

To put this branch (with changes) on the server, we simply need to click “Publish Branch”. Navigating to the repo page, we will see a suggestion about a new branch
At which point the process of creating a PR is identical to the previous case.

12.6.4 Julia Package Case

One special case is when the repo in question is actually a Julia project or package. We cover that (along with package workflow in general) in the testing lecture.

12.7 Additional Resources and Troubleshooting

You may want to go beyond the scope of this tutorial when working with GitHub. For example, perhaps you run into a bug, or you’re working with a setup that doesn’t have GitHub Desktop installed.

Here are some resources to help

- Kate Hudson’s excellent git flight rules, which is a near-exhaustive list of situations you could encounter, and command-line fixes.
- The GitHub Learning Lab, an interactive sandbox environment for git.
- The docs for forking on GitHub Desktop and the GitHub Website.

12.7.1 Command-Line Basics

Git also comes with a set of command-line tools. They’re optional, but many people like using them.

Furthermore, in some environments (e.g. JupyterHub installations) you may only have access to the command line.

- On Windows, downloading git will have installed a program called git bash, which installs these tools along with a general Linux-style shell.
- On Linux/MacOS, these tools are integrated into your usual terminal.

To open the terminal in a directory, either right click and hit “open git bash” (in Windows), or use Linux commands like cd and ls to navigate.
See here for a short introduction to the command line.

As above, you can clone by grabbing the repo URL (say, GitHub's site-policy repo) and running `git clone https://github.com/github/site-policy.git`.

This won’t be connected to your GitHub Desktop, so you’d need to use it manually (File => Add Local Repository) or drag-and-drop from the file explorer onto the GitHub Desktop.

From here, you can get the latest files on the server by `cd`-ing into the directory and running `git pull`.

When you pull from the server, it will never overwrite your modified files, so it is impossible to lose local changes.

Instead, to do a hard reset of all files and overwrite any of your local changes, you can run `git reset --hard origin/master`.

## 12.8 Exercises

### 12.8.1 Exercise 1a

Follow the instructions to create a new repository for one of your GitHub accounts. In this repository:

- Take the code from one of your previous assignments, such as Newton’s method in Introductory Examples (either as a `.jl` file or a Jupyter notebook).
- Put in a `README.md` with some text.
- Put in a `.gitignore` file, ignoring the Jupyter files `.ipynb_checkpoints` and the project files, `.projects`.

### 12.8.2 Exercise 1b

Pair-up with another student who has done Exercise 1a and find out their GitHub ID, and each do the following:

- Add the GitHub ID as a collaborators on your repository.
• Clone the repositories to your local desktop.
• Assign each other an issue.
• Submit a commit from GitHub Desktop which references the issue by number.
• Comment on the commits.
• Ensure you can run their code without any modifications.

12.8.3 Exercise 1c

Pair-wise with the results of Exercise 1b examine a merge-conflict by editing the README.md file for your repository that you have both setup as collaborators.

Start by ensuring there are multiple lines in the file so that some changes may have conflicts, and some may not.

• Clone the repository to your local desktops.
• Modify different lines of code in the file and both commit and push to the server (prior to pulling from each other)—and see how it merges things “automatically”.
• Modify the same line of code in the file, and deal with the merge conflict.

12.8.4 Exercise 2a

Just using GitHub’s web interface, submit a Pull Request for a simple change of documentation to a public repository.

The easiest may be to submit a PR for a typo in the source repository for these notes, i.e. https://github.com/QuantEcon/lecture-source-jl.

Note: The source for that repository is in .rst files, but you should be able to find spelling mistakes/etc. without much effort.

12.8.5 Exercise 2b

Following the instructions for forking and cloning a public repository to your local desktop, submit a Pull Request to a public repository.

Again, you could submit it for a typo in the source repository for these notes, i.e. https://github.com/QuantEcon/lecture-source-jl, but you are also encouraged to instead look for a small change that could help the documentation in another repository.

If you are ambitious, then go to the Exercise Solutions for one of the Exercises in these lecture notes and submit a PR for your own modified version (if you think it is an improvement!).
Chapter 13

Packages, Testing, and Continuous Integration

13.1 Contents

- Project Setup 13.2
- Project Structure 13.3
- Project Workflow 13.4
- Unit Testing 13.5
- Continuous Integration with Travis 13.6
- Code Coverage 13.7
- Pull Requests to External Julia Projects 13.8
- Benchmarking 13.9
- Additional Notes 13.10
- Review 13.11
- Exercises 13.12

A complex system that works is invariably found to have evolved from a simple system that worked. The inverse proposition also appears to be true: A complex system designed from scratch never works and cannot be made to work. You have to start over, beginning with a working simple system – Gall’s Law.

Co-authored with Arnav Sood

This lecture discusses structuring a project as a Julia module, and testing it with tools from GitHub.

Benefits include

- Specifying dependencies (and their versions) so that your project works across Julia setups and over time.
- Being able to load your project’s functions from outside without copy/pasting.
- Writing tests that run locally, and automatically on the GitHub server.
- Having GitHub test your project across operating systems, Julia versions, etc.


13.2 Project Setup

13.2.1 Account Setup

Travis CI

As we’ll see later, Travis is a service that automatically tests your project on the GitHub server.

First, we need to make sure that your GitHub account is set up with Travis CI and Codecov. As a reminder, make sure you signed up for the GitHub Student Developer Pack or Academic Plan if eligible.

Navigate to the travis-ci.com website and click “sign up with GitHub” – supply your credentials.

If you get stuck, see the Travis tutorial.

Codecov

Codecov is a service that tells you how comprehensive your tests are (i.e., how much of your code is actually tested).

To sign up, visit the Codecov website, and click “sign up”

Next, click “add a repository” and enable private scope (this allows Codecov to service your private projects).

The result should be
This is all we need for now.

13.2.2 Julia Setup

In [1]: using InstantiateFromURL
   # optionally add arguments to force installation: instantiate = true,
   # precompile = true
   github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

Note: Before these steps, make sure that you’ve either completed the version control lecture or run.

git config --global user.name "Your Name"

Note: Throughout this lecture, important points and sequential workflow steps are listed as bullets.

To set up a project on Julia:

- Load the PkgTemplates package.

using PkgTemplates

- Create a template for your project.

This specifies metadata like the license we’ll be using (MIT by default), the location (~/.julia/dev by default), etc.

ourTemplate = Template(;user="quanteconuser", plugins = [TravisCI(), Codecov()],

Note: Make sure you replace the quanteconuser with your GitHub ID.

- Create a specific project based off this template
generate("ExamplePackage.jl", ourTemplate)

If we navigate to the package directory, we should see something like the following.

As a reminder, the location of your `.julia` folder can be found by running `DEPOT_PATH[1]` in a REPL.

**Note:** On Mac, this may be hidden; you can either start a terminal, `cd ~` and then `cd .julia`, or make hidden files visible in the Finder.

### 13.2.3 Adding a Project to Git

The next step is to add this project to Git version control.

- Open the repository screen in your account as discussed previously.

We’ll want the following settings

In particular

- The repo you create should have the same name as the project we added.
- We should leave the boxes unchecked for the README.md, LICENSE, and .gitignore, since these are handled by PkgTemplates.

Then,
13.2. PROJECT SETUP

- Drag and drop your folder from your ~/.julia/dev directory to GitHub Desktop.
- Click the “publish branch” button to upload your files to GitHub.

If you navigate to your git repo (ours is here), you should see something like

![GitHub Desktop](image)

**Note:** Be sure that you don’t separately clone the repo you just added to another location (i.e., to your desktop).

A key note is that you have some set of files on your local machine (here in ~/.julia/dev/ExamplePackage.jl) and git is plugged into those files.

For convenience, you might want to create a shortcut to that location somewhere accessible.

### 13.2.4 Adding a Project to the Julia Package Manager

We also want Julia’s package manager to be aware of the project.

- Open a REPL in the newly created project directory, either by noting the path printed above, or by running the following in a REPL.

```julia
cd(joinpath(DEPOT_PATH[1], "dev", "ExamplePackage"))
```

Note the lack of .jl!

- Run the following

```julia
] activate
```

to get into the main Julia environment (more on environments in the second half of this lecture).

- And run

```julia
] dev .
```
to add the package.

You can see the change reflected in our default package list by running
For more on the package mode, see the tools and editors lecture.

### 13.2.5 Using the Package Manager

Now, from any Julia terminal in the future, we can run

```julia
using ExamplePackage
```

To use its exported functions.

We can also get the path to this by running

```julia
using ExamplePackage
pathof(ExamplePackage) # returns path to src/ExamplePackage.jl
```

### 13.3 Project Structure

Let’s unpack the structure of the generated project

- The first directory, `.git`, holds the version control information.
- The `src` directory contains the project’s source code – it should contain only one file (`ExamplePackage.jl`), which reads

```julia
module ExamplePackage

greet() = print("Hello World!")

end # module
```

- Likewise, the `test` directory should have only one file (`runtests.jl`), which reads

```julia
using ExamplePackage
using Test

@testset "ExamplePackage.jl" begin
    # Write your own tests here.
end
```

In particular, the workflow is to export objects we want to test (`using ExamplePackage`), and test them using Julia's `Test` module.

The other important text files for now are

- `Project.toml` and `Manifest.toml`, which contain dependency information.

In particular, the `Project.toml` contains a list of dependencies, and the `Manifest.toml` specifies their exact versions and sub-dependencies.

- The `.gitignore` file (which may display as an untitled file), which contains files and paths for `git` to ignore.
13.4 Project Workflow

13.4.1 Dependency Management

Environments

As before, the .toml files define an environment for our project, or a set of files which represent the dependency information.

The actual files are written in the TOML language, which is a lightweight format to specify configuration options.

This information is the name of every package we depend on, along with the exact versions of those packages.

This information (in practice, the result of package operations we execute) will be reflected in our ExamplePackage.jl directory’s TOML, once that environment is activated (selected).

This allows us to share the project with others, who can exactly reproduce the state used to build and test it.

See the Pkg3 docs for more information.

Pkg Operations

For now, let’s just try adding a dependency

- Activate the package environment (to be run from the base, v1.1 environment)

] activate ExamplePackage

This tells Julia to write the results of package operations to ExampleProject’s TOML, and use the versions of packages specified there.

Note that the base environment isn’t special, except that it’s what’s loaded by a freshly-started REPL or Jupyter notebook.

- Add a package

] add Expectations

We can track changes in the TOML, as before.

Here’s the Manifest.toml
We can also run other operations, like ] up, ] precompile, etc.
Package operations are listed in detail in the tools and editors lecture.
Recall that, to quit the active environment and return to the base (v1.1), simply run

] activate

13.4.2 Writing Code

The basic idea is to work in tests/runtests.jl, while reproducible functions should go in the src/ExamplePackage.jl.
For example, let’s say we add Distributions.jl

] activate ExamplePackage

] add Distributions

and edit the source (paste this into the file itself) to read as follows

```julia
module ExamplePackage

greet() = print("Hello World!")

using Expectations, Distributions

function foo(μ = 1., σ = 2.)
    d = Normal(μ, σ)
    E = expectation(d)
    return E(x -> sin(x))
end

export foo

end # module
```
Let’s try calling this

```julia
] activate

using ExamplePackage
ExamplePackage.greet()

foo() # exported, so don’t need to qualify the namespace
```

**Note:** You may need to quit your REPL and load the package again.

### 13.4.3 Jupyter Workflow

We can also work with the package from a Jupyter notebook.

Let’s create a new output directory in our project, and run `jupyter lab` from it.

Create a new notebook `output.ipynb`

![Jupyter Notebook](image)

From here, we can use our package’s functions as we would functions from other packages. This lets us produce neat output documents, without pasting the whole codebase.

We can also run package operations inside the notebook

![Jupyter Notebook with Code](image)

The change will be reflected in the `Project.toml` file.

Note that, as usual, we had to first activate `ExamplePackage` first before making our dependency changes

```toml
name = "ExamplePackage"
uuid = "f85830d0-e1f0-11e8-2fad-8762162ab251"
authors = ["QuantEcon User <quanteconuser@gmail.com>"]
```
version = "0.1.0"

[deps]
Distributions = "31c24e10-a181-5473-b8eb-7969acd0382f"
Expectations = "2fe49d83-0758-5602-8f54-1f90ad0d522b"
Parameters = "d96e819e-fc66-5662-9728-84c9c7592b0a"

[extras]
Test = "8dfed614-e22c-5e08-85e1-65c5234f0b40"

(targets)
test = ["Test"]

There will also be changes in the Manifest as well.
Be sure to add output/.ipynb_checkpoints to your .gitignore file, so that's not checked in.
Make sure you've activated the project environment (] activate ExamplePackage) before you try to propagate changes.

13.4.4 Collaborative Work

For someone else to get the package, they simply need to

- Run the following command

] dev https://github.com/quanteconuser/ExamplePackage.jl.git

This will place the repository inside their ~/.julia/dev folder.
- Drag-and-drop the folder to GitHub desktop in the usual way.
Recall that the path to your ~/.julia folder is

In [2]: DEPOT_PATH[1]

Out[2]: "/home/ubuntu/.julia"

They can then collaborate as they would on other git repositories.
In particular, they can run

] activate ExamplePackage

to load the list of dependencies, and

] instantiate

to make sure they are installed on the local machine.
13.5 Unit Testing

It’s important to make sure that your code is well-tested.

There are a few different kinds of test, each with different purposes

- *Unit testing* makes sure that individual pieces of a project work as expected.
- *Integration testing* makes sure that they fit together as expected.
- *Regression testing* makes sure that behavior is unchanged over time.

In this lecture, we’ll focus on unit testing.

In general, well-written unit tests (which also guard against regression, for example by comparing function output to hardcoded values) are sufficient for most small projects.

13.5.1 The Test Module

Julia provides testing features through a built-in package called Test, which we get by using Test.

The basic object is the macro `@test`

```julia
In [3]: using Test
   @test 1 == 1
   @test 1 ≈ 1
```

**Out[3]: Test Passed**

Tests will pass if the condition is `true`, or fail otherwise.

If a test is failing, we should flag it with `@test_broken` as below

```julia
In [4]: @test_broken 1 == 2
```

**Out[4]: Test Broken**

Expression: 1 == 2

This way, we still have access to information about the test, instead of just deleting it or commenting it out.

There are other test macros, that check for things like error handling and type-stability. Advanced users can check the Julia docs.

13.5.2 Example

Let’s add some unit tests for the `foo()` function we defined earlier.

Our `tests/runtests.jl` file should look like this.

As before, this should be pasted into the file directly

```julia
using ExamplePackage
using Test
```
@test foo() ≈ 0.11388071406436832
@test foo(1, 1.5) ≈ 0.2731856314283442
@test_broken foo(1, 0) # tells us this is broken

And run it by typing ] test into an activated REPL (i.e., a REPL where you’ve run ]
activate ExamplePackage).

### 13.5.3 Test Sets

By default, the runtests.jl folder starts off with a @testset.
This is useful for organizing different batches of tests, but for now we can simply ignore it.
To learn more about test sets, see the docs.

### 13.5.4 Running Tests

There are a few different ways to run the tests for your package.

- Run the actual runtests.jl, say by hitting shift-enter on it in Atom.
- From a fresh (v1.1) REPL, run ] test ExamplePackage.
- From an activated (ExamplePackage) REPL, simply run ] test (recall that you can
  activate with ] activate ExamplePackage).

### 13.6 Continuous Integration with Travis

#### 13.6.1 Setup

By default, Travis should have access to all your repositories and deploy automatically.
This includes private repos if you’re on a student developer pack or an academic plan (Travis
detects this automatically).
To change this, go to “settings” under your GitHub profile
Click “Applications,” then “Travis CI,” then “Configure,” and choose the repos you want to be tracked.

### 13.6.2 Build Options

By default, Travis will compile and test your project (i.e., “build” it) for new commits and PRs for every tracked repo with a `.travis.yml` file.

We can see ours by opening it in Atom

```yaml
# Documentation: http://docs.travis-ci.com/user/languages/julia/
language: julia
os:
  - linux
  - osx
julia:
  - 1.1
  - nightly
matrix:
allow_failures:
  - julia: nightly
fast_finish: true
notifications:
email: false
after_success:
  - julia -e 'using Pkg; Pkg.add("Coverage"); using Coverage; Codecov.submit(process_folder())'
```
This is telling Travis to build the project in Julia, on OSX and Linux, using Julia v1.1 and the latest development build (“nightly”).

It also says that if the nightly version doesn’t work, that shouldn’t register as a failure.

**Note** You won’t need OSX unless you’re building something Mac-specific, like iOS or Swift. You can delete those lines to speed up the build, likewise for the nightly Julia version.

### 13.6.3 Working with Builds

As above, builds are triggered whenever we push changes or open a pull request.

For example, if we push our changes to the server and then click the Travis badge (the one which says “build”) on the README, we should see something like

![Travis CI Dashboard](image)

Note that you may need to wait a bit and/or refresh your browser.

This gives us an overview of all the builds running for that commit.

To inspect a build more closely (say, if it fails), we can click on it and expand the log options.
Note that the build times here aren’t informative, because we can’t generally control the hardware to which our job is allocated.

We can also cancel specific jobs, either from their specific pages or by clicking the grey “x” button on the dashboard.

Lastly, we can trigger builds manually (without a new commit or PR) from the Travis overview.

To commit without triggering a build, simply add “[ci skip]” somewhere inside the commit message.

13.6.4 Travis and Pull Requests

One key feature of Travis is the ability to see at-a-glance whether PRs pass tests before merging them.

This happens automatically when Travis is enabled on a repository.

For an example of this feature, see this PR in the Games.jl repository.
13.7 Code Coverage

Beyond the success or failure of our test suite, we also want to know how much of our code the tests cover.

The tool we use to do this is called **Codecov**.

### 13.7.1 Setup

You’ll find that Codecov is automatically enabled for public repos with Travis.

For private ones, you’ll need to first get an access token.

Add private scope in the Codecov website, just like we did for Travis.

Navigate to the repo settings page (i.e., [https://codecov.io/gh/quanteconuser/ExamplePackage](https://codecov.io/gh/quanteconuser/ExamplePackage) for our repo) and copy the token.

Next, go to your Travis settings and add an environment variable as below

![Travis CI Settings](image)

### 13.7.2 Interpreting Results

Click the Codecov badge to see the build page for your project.

This shows us that our tests cover 50% of our functions in `src/`

**Note:** To get a more detailed view, we can click the `src/` and the resultant filename.

**Note:** Codecov may take a few minutes to run for the first time
This shows us precisely which methods (and parts of methods) are untested.

### 13.8 Pull Requests to External Julia Projects

As mentioned in version control, sometimes we’ll want to work on external repos that are also Julia projects.

- `] dev` the git URL (or package name, if the project is a registered Julia package), which will both clone the git repo to `~/.julia/dev` and sync it with the Julia package manager.

For example, running

```julia
] dev Expectations
```

will clone the repo `https://github.com/quantecon/Expectations.jl` to `~/.julia/dev/Expectations`.

Make sure you do this from the base Julia environment (i.e., after running `] activate` without arguments).

As a reminder, you can find the location of your `~/.julia` folder (called the “user depot”), by running

```
In [5]: DEPOT_PATH[1]
```

```
Out[5]: "/home/ubuntu/.julia"
```

The `] dev` command will also add the target to the package manager, so that whenever we run `using Expectations`, Julia will load our cloned copy from that location

```julia
using Expectations
pathof(Expectations)  # points to our git clone
```
• Drag that folder to GitHub Desktop.
• The next step is to fork the original (external) package from its website (i.e., https://github.com/quantecon/Expectations.jl) to your account (https://github.com/quanteconuser/Expectations.jl in our case).

![GitHub Desktop](image1)

• Edit the settings in GitHub Desktop (from the “Repository” dropdown) to reflect the new URL.

![Repository settings](image2)

Here, we’d change the highlighted text to read `quanteconuser`, or whatever our GitHub ID is.

• If you make some changes in a text editor and return to GitHub Desktop, you’ll see something like.

Note: As before, we’re editing the files directly in `~/.julia/dev`, as opposed to cloning the repo again.

![GitHub Desktop](image3)

Here, for example, we’re revising the README.

• Clicking “commit to master” (recall that the checkboxes next to each file indicate
13.8. PULL REQUESTS TO EXTERNAL JULIA PROJECTS

whether it’s to be committed) and then pushing (e.g., hitting “push” under the “Repository” dropdown) will add the committed changes to your account.

To confirm this, we can check the history on our account here; for more on working with git repositories, see the version control lecture.

The green check mark indicates that Travis tests passed for this commit.

- Clicking “new pull request” from the pull requests tab will show us a snapshot of the changes, and let us create a pull request for project maintainers to review and approve.

For more on PRs, see the relevant section of the version control lecture.

For more on forking, see the docs on GitHub Desktop and the GitHub Website.

13.8.1 Case with Write Access

If you have write access to the repo, we can skip the preceding steps about forking and changing the URL.

You can use ] dev on a package name or the URL of the package

] dev Expectations

or ] dev https://github.com/quanteconuser/Expectations.jl.git as an example for an unreleased package by URL.

Which will again clone the repo to ~/.julia/dev, and use it as a Julia package.

using Expectations
pathof(Expectations) # points to our git clone
Next, drag that folder to GitHub Desktop as before.

Then, in order to work with the package locally, all we need to do is open the 
~/.julia/dev/Expectations in a text editor (like Atom).

From here, we can edit this package just like we created it ourselves and use GitHub Desktop to track versions of our package files (say, after ] up, or editing source code, ] add Package, etc.).

13.8.2 Removing a Julia Package

To “un-dev” a Julia package (say, if we want to use our old Expectations.jl), you can simply run

] free Expectations

To delete it entirely, simply run

] rm Expectations

From a REPL where that package is in the active environment.

13.9 Benchmarking

Another goal of testing is to make sure that code doesn’t slow down significantly from one version to the next.

We can do this using tools provided by the BenchmarkTools.jl package.

See the need for speed lecture for more details.

13.10 Additional Notes

- The JuliaCI organization provides more Julia utilities for continuous integration and testing.
13.11 Review

To review the workflow for creating, versioning, and testing a new project end-to-end.

1. Create the local package directory using the `PkgTemplates.jl`.

2. Add that package to the Julia package manager, by opening a Julia REPL in the

   `~/.julia/dev/ExamplePackage.jl` directory, making sure the active environment is the default one (`v1.1`), and hitting ] dev ..

3. Drag-and-drop that folder to GitHub Desktop.

4. Create an empty repository with the same name on the GitHub server.

5. Push from GitHub Desktop to the server.

6. Open the original project folder (e.g., `~/.julia/dev/ExamplePackage.jl`) in Atom.

7. Make changes, test, iterate on it, etc. As a rule, functions like should live in the `src/` directory once they’re stable, and you should export them from that file with `export func1, func2`. This will export all methods of `func1`, `func2`, etc.

8. Commit them in GitHub Desktop as you go (i.e., you can and should use version control to track intermediate states).

9. Push to the server, and see the Travis and Codecov results (note that these may take a few minutes the first time).

13.12 Exercises

13.12.1 Exercise 1

Following the instructions for a new project, create a new package on your github account called `NewtonsMethod.jl`.

In this package, you should create a simple package to do Newton’s Method using the code you did in the Newton’s method exercise in Introductory Examples.

In particular, within your package you should have two functions

- `newtonroot(f, f'; x₀, tol = 1E-7, maxiter = 1000)`
- `newtonroot(f; x₀, tol = 1E-7, maxiter = 1000)`

Where the second function uses Automatic Differentiation to call the first.

The package should include

- implementations of those functions in the `/src` directory
- comprehensive set of tests
- project and manifest files to replicate your development environment
- automated running of the tests with Travis CI in GitHub

For the tests, you should have at the very minimum

- a way to handle non-convergence (e.g. return back `nothing` as discussed in error handling)
• several `@test` for the root of a known function, given the $f$ and analytical $f'$ derivatives
• tests of those roots using the automatic differentiation version of the function
• test of finding those roots with a `BigFloat` and not just a `Float64`
• test of non-convergence for a function without a root (e.g. $f(x) = 2 + x^2$)
• test to ensure that the `maxiter` is working (e.g. what happens if you call `maxiter = 5`)
• test to ensure that `tol` is working

And anything else you can think of. You should be able to run `] test` for the project to check that the test-suite is running, and then ensure that it is running automatically on Travis CI.

Push a commit to the repository which breaks one of the tests and see what the Travis CI reports after running the build.

13.12.2 Exercise 2

Watch the youtube video Developing Julia Packages from Chris Rackauckas. The demonstration goes through many of the same concepts as this lecture, but with more background in test-driven development and providing more details for open-source projects.
Chapter 14

The Need for Speed

14.1 Contents

- Overview 14.2
- Understanding Multiple Dispatch in Julia 14.3
- Foundations 14.4
- JIT Compilation in Julia 14.5
- Fast and Slow Julia Code 14.6
- Further Comments 14.7

14.2 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.
14.2.1 Requirements

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

14.2.2 Setup

**In [1]:** `using InstantiateFromURL`
   # optionally add arguments to force installation: instantiate = true, 
   precompile = true
   github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

**In [2]:** `using LinearAlgebra, Statistics`

14.3 Understanding Multiple Dispatch in Julia

This section provides more background on how methods, functions, and types are connected.

14.3.1 Methods and Functions

The precise data type is important, for reasons of both efficiency and mathematical correctness.

For example consider `1 + 1` vs. `1.0 + 1.0` or `[1 0] + [0 1]`.

On 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 compiler inspects the type of data to be acted on and hands it out to the appropriate method.

This process is called **multiple dispatch**.

Like all “infix” operators, `1 + 1` has the alternative syntax `+(1, 1)`

**In [3]:** `+(1, 1)`

**Out[3]:** `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

**In [4]:** `x, y = 1.0, 1.0
   @which +(x, y)`

**Out[4]:** `+(x::Float64, y::Float64) in Base at float.jl:401`
We see that the operation is sent to the + method that specializes in adding floating point numbers.

Here’s the integer case

In [5]: x, y = 1, 1
   @which +(x, y)

Out[5]: +(x::T, y::T) where T<:Union{Int128, Int16, Int32, Int64, Int8, UInt128, ...
    -UInt16, UInt32, UInt64, UInt8} in Base at int.jl:53

This output says that the call has been dispatched to the + method responsible for handling integer values.

(We’ll learn more about the details of this syntax below)

Here’s another example, with complex numbers

In [6]: x, y = 1.0 + 1.0im, 1.0 + 1.0im
   @which +(x, y)

Out[6]: +(z::Complex, w::Complex) in Base at complex.jl:275

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

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 (i.e. \(100 + "100"\) is not valid syntax).

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

In [7]: import Base: +  # enables adding methods to the + function
   +(x::Integer, y::String) = x + parse(Int, y)
  @show +(100, "100")
  @show 100 + "100";  # equivalent

    100 + "100" = 200
    100 + "100" = 200

14.3.2 Understanding the Compilation Process

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 `ComplexF32` then the immediate parents are `AbstractFloat` and `Number` respectively.

```
In [8]: supertype(Float64)
Out[8]: AbstractFloat

In [9]: supertype(ComplexF32)
Out[9]: 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 (since we only added the + for adding `Integer` and `String` above):

```
In [10]:@show (typeof(100.0) <: Integer) == false
100.0 + "100"

    (typeof(100.0) <: Integer) == false = true

    MethodError: no method matching +(::Float64, ::String)
    Closest candidates are:
    +(::Any, ::Any, !Matched::Any, !Matched::Any...) at operators.jl:529
    +(::Float64, !Matched::Float64) at float.jl:401
    +(::AbstractFloat, !Matched::Bool) at bool.jl:106

    Stacktrace:
    [1] top-level scope at In[10]:2
```
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. \((\text{Float64}, \text{Int64})\) pairs
2. \((\text{Number}, \text{Number})\) pairs

and you call \( f(0.5, 1) \) 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 q(x)  # or q(x::Any)
    println("Default (Any) method invoked")
end

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

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

```
Out[11]: q (generic function with 3 methods)
```

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

```julia
In [12]: q(3)

Integer method invoked

In [13]: q(3.0)

Number method invoked

In [14]: q("foo")

Default (Any) method invoked
```

Since `typeof(3) <: \text{Int64} <: \text{Integer} <: \text{Number}` , the call `q(3)` proceeds up the tree to `\text{Integer}` and invokes `q(x::\text{Integer})`.

On the other hand, `3.0` is a `\text{Float64}` , which is not a subtype of `\text{Integer}`.

Hence the call `q(3.0)` continues up to `q(x::\text{Number})`.

Finally, `q("foo")` is handled by the function operating on `\text{Any}` , since `\text{String}` is not a subtype of `\text{Number}` or `\text{Integer}`.
14.3.3 Analyzing Function Return Types

For the most part, time spent “optimizing” Julia code to run faster is about ensuring the compiler can correctly deduce types for all functions.

The macro `@code_warntype` gives us a hint

```
In [15]: x = [1, 2, 3]
f(x) = 2x
@code_warntype f(x)
```

```
Variables
#self#:Core.Compiler.Const(f, false)
x::Array{Int64,1}

Body::Array{Int64,1}
1 ─ %1 = (2 * x)::Array{Int64,1}
   └── return %1
```

The `@code_warntype` macro compiles `f(x)` using the type of `x` as an example – i.e., the `[1, 2, 3]` is used as a prototype for analyzing the compilation, rather than simply calculating the value.

Here, the `Body::Array{Int64,1}` tells us the type of the return value of the function, when called with types like `[1, 2, 3]`, is always a vector of integers.

In contrast, consider a function potentially returning `nothing`, as in this lecture

```
In [16]: f(x) = x > 0.0 ? x : nothing
@code_warntype f(1)
```

```
Variables
#self#:Core.Compiler.Const(f, false)
x::Int64

Body::Union{Nothing, Int64}
1 ─ %1 = (x > 0.0)::Bool
   └── goto #3 if not %1
2 ─ return x
3 ─ return Main.nothing
```

This states that the compiler determines the return type when called with an integer (like 1) could be one of two different types, `Body::Union{Nothing, Int64}`.

A final example is a variation on the above, which returns the maximum of `x` and 0.

```
In [17]: f(x) = x > 0.0 ? x : 0.0
@code_warntype f(1)
```

```
Variables
#self#:Core.Compiler.Const(f, false)
x::Int64

Body::Union{Float64, Int64}
```
14.4 **FOUNDATIONS**

1 - \%1 = (x > 0.0)::Bool
   \[\text{goto } #3 \text{ if not } \%1\]
2 - \text{return } x
3 - \text{return } 0.0

Which shows that, when called with an integer, the type could be that integer or the floating point \(0.0\).

On the other hand, if we use change the function to return \(0\) if \(x \leq 0\), it is type-unstable with floating point.

In [18]: \(f(x) = x > 0.0 ? x : 0\)
   
   @code_warntype \(f(1.0)\)

   Variables
   #self#::Core.Compiler.Const(f, false)
x::Float64

   Body::Union{Float64, Int64}
   1 - \%1 = (x > 0.0)::Bool
   \[\text{goto } #3 \text{ if not } \%1\]
2 - \text{return } x
3 - \text{return } 0

The solution is to use the \texttt{zero(x)} function which returns the additive identity element of type \(x\).

On the other hand, if we change the function to return \(0\) if \(x \leq 0\), it is type-unstable with floating point.

In [19]: @show zero(2.3)
   @show zero(4)
   @show zero(2.0 + 3im)

   \(f(x) = x > 0.0 ? x : \text{zero}(x)\)
   @code_warntype \(f(1.0)\)

   zero(2.3) = 0.0
zero(4) = 0
zero(2.0 + 3im) = 0.0 + 0.0im

Variables
#self#::Core.Compiler.Const(f, false)
x::Float64

Body::Float64
1 - \%1 = (x > 0.0)::Bool
   \[\text{goto } #3 \text{ if not } \%1\]
2 - \text{return } x
3 - \%4 = Main.zero(x)::Core.Compiler.Const(0.0, false)
   \[\text{return } \%4\]

14.4 **Foundations**

Let’s think about how quickly code runs, taking as given
We’ll start by discussing the kinds of instructions that machines understand.

### 14.4.1 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 $2a + 8b$

```assembly
pushq  %rbp
movq  %rsp, %rbp
addq  %rdi, %rdi
leaq  (%rdi,%rsi,8), %rax
popq  %rbp
retq
nopl  (%rax)
```

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.

```python
In [20]: function f(a, b)
   ...:     y = 2a + 8b
   ...:     return y
   ...: end
Out[20]: f (generic function with 2 methods)
```

or Python

```python
def f(a, b):
    y = 2 * a + 8 * b
    return y
```

or even C
```c
int f(int a, int b) {
    int y = 2 * a + 8 * b;
    return y;
}
```

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?

### 14.4.2 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

### 14.5 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.

#### 14.5.1 An Example

Consider the function

```julia
In [21]: function f(a, b)
    y = (a + 8b)^2
    return 7y
end
```

**Out[21]:** `f (generic function with 2 methods)`

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
14.5. JIT COMPILATION IN JULIA

- 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

In [22]: `@code_native f(1, 2)`

```
.text
; @ In[21]:2 within 'f'
  leaq (%rdi,%rsi,8), %rcx
; | @ In[21]:2 within '+'
  imulq %rcx, %rcx
; | @ intfuncs.jl:261 within 'literal_pow'
  leaq (%rcx,8), %rax
  subq %rcx, %rax
  retq
  nopw %cs:(%rax,%rax)
  nop
; | @ int.jl:54 within '*'
    imulq %rcx, %rcx
    leaq (%rcx,8), %rax
    subq %rcx, %rax
    retq
    nopw %cs:(%rax,%rax)
    nop
; | @ In[21]:3 within 'f'
    movabsq $139778094801248, %rax # imm = 0x7F20CA494820
    vaddsd %xmm0, %xmm1, %xmm2
    vaddsd %xmm0, %xmm1, %xmm2
    vmulsd (%rax), %xmm0, %xmm0
    movabsq $139778094801240, %rax # imm = 0x7F20CA494818
    vaddsd %xmm0, %xmm1, %xmm2
    vaddsd %xmm0, %xmm1, %xmm2
    vmulsd (%rax), %xmm0, %xmm0
; | @ promotion.jl:312 within '*' @ float.jl:405
  vmulsd (%rax), %xmm0, %xmm0
; | @ intfuncs.jl:261 within 'literal_pow'
  vmulsd %xmm0, %xmm0, %xmm0
  movabsq $139778094801248, %rax # imm = 0x7F20CA494820
; | @ float.jl:405 within 'f'
  vmulsd (%rax), %xmm0, %xmm0
  vaddsd %xmm0, %xmm1, %xmm2
  vaddsd %xmm0, %xmm1, %xmm2
; | @ float.jl:405 within '+'
  vmulsd (%rax), %xmm0, %xmm0
  vaddsd %xmm0, %xmm1, %xmm2
  vaddsd %xmm0, %xmm1, %xmm2
; | @ intfuncs.jl:261 within 'literal_pow'
  vmulsd (%rax), %xmm0, %xmm0
  movabsq $139778094801240, %rax # imm = 0x7F20CA494818
; | @ promotion.jl:312 within '+'
  vmulsd (%rax), %xmm0, %xmm0
  vaddsd %xmm0, %xmm1, %xmm2
  vaddsd %xmm0, %xmm1, %xmm2
; | @ int.jl:54 within '*'
    imulq %rcx, %rcx
    leaq (%rcx,8), %rax
    subq %rcx, %rax
    retq
    nopw %cs:(%rax,%rax)
    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.

In [23]: `@code_native f(1.0, 2.0)`

```
.text
; @ In[21]:2 within 'f'
  movabsq $139778094801240, %rax # imm = 0x7F20CA494818
; | @ promotion.jl:312 within '*' @ float.jl:405
  vmulsd (%rax), %xmm0, %xmm0
; | @ float.jl:405 within '+'
  vaddsd %xmm0, %xmm1, %xmm0
; | @ float.jl:405 within '+'
  vaddsd %xmm0, %xmm1, %xmm2
; | @ intfuncs.jl:261 within 'literal_pow'
  vmulsd %xmm0, %xmm0, %xmm0
  movabsq $139778094801248, %rax # imm = 0x7F20CA494820
; | @ promotion.jl:312 within '+'
  vmulsd (%rax), %xmm0, %xmm0
; | @ float.jl:405 within '+'
  vaddsd %xmm0, %xmm1, %xmm2
  vaddsd %xmm0, %xmm1, %xmm2
; | @ intfuncs.jl:261 within 'literal_pow'
  vmulsd (%rax), %xmm0, %xmm0
  movabsq $139778094801240, %rax # imm = 0x7F20CA494818
; | @ int.jl:54 within '*'
    imulq %rcx, %rcx
    leaq (%rcx,8), %rax
    subq %rcx, %rax
    retq
    nopw %cs:(%rax,%rax)
    nop
; |
```
Subsequent calls using either floats or integers are now routed to the appropriate compiled code.

### 14.5.2 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.

### 14.6 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.

#### 14.6.1 BenchmarkTools

The main Julia package for benchmarking is BenchmarkTools.jl. Below, we'll use the @btime macro it exports to evaluate the performance of Julia code.

As mentioned in an earlier lecture, we can also save benchmark results to a file and guard against performance regressions in code.

For more, see the package docs.

#### 14.6.2 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
In [24]: b = 1.0
function g(a)
global b
for i = 1:1_000_000
    tmp = a + b
end
end
```

Out[24]: g (generic function with 1 method)

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

```julia
In [25]: using BenchmarkTools
@btime g(1.0)
30.018 ms (2000000 allocations: 30.52 MiB)
```

If you look at the corresponding machine code you will see that it's a mess.

```asm
In [26]: @code_native g(1.0)
```

```
.text
@ In[24]:3 within `g'
pushq %rbp
movq %rsp, %rbp
pushq %r15
pushq %r14
pushq %r13
pushq %r12
pushq %rbx
andq $-32, %rsp
subq $128, %rsp
vmovsd %xmm0, 24(%rsp)
vxorps %xmm0, %xmm0, %xmm0
vmovaps %ymm0, 32(%rsp)
movq %fs:0, %rax
movq $8, 32(%rsp)
movq -15712(%rax), %rcx
movq %rcx, 40(%rsp)
lea 32(%rsp), %rcx
movq %rcx, -15712(%rax)
lea -15712(%rax), %r12
movl $1000000, %ebx  # imm = 0xF4240
```
CHAPTER 14. THE NEED FOR SPEED

If we eliminate the global variable like so

In [27]: function g(a, b)
   for i ∼ 1:1_000_000
      tmp = a + b
   end
end

Out[27]: g (generic function with 2 methods)

then execution speed improves dramatically

In [28]: @btime g(1.0, 1.0)
Note that the second run was dramatically faster than the first. That’s 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.

```
In [29]: @code_native g(1.0, 1.0)

.text
; r @ In[27]:2 within `g'
  retq
  nopw %cs:(%rax,%rax)
  nopl (%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`.

```
In [30]: const b_const = 1.0

function g(a)
  global b_const
  for i ⊕ 1:1_000_000
    tmp = a + b_const
  end
end
```

```
Out[30]: g (generic function with 2 methods)
```

Now the compiler can again generate efficient machine code.

We’ll leave you to experiment with it.

### 14.6.3 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.

Let’s experiment, using, respectively,

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

As we’ll see, the last of these options gives us the best performance, while still maintaining significant flexibility.

Here’s the untyped case

In [31]: struct Foo_generic
    
    a
    
    end

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

In [32]: struct Foo_abstract
    
    a::Real
    
    end

Finally, here’s the parametrically typed case

In [33]: struct Foo_concrete{T <: Real}
    
    a::T
    
    end

Now we generate instances

In [34]: fg = Foo_generic(1.0)
    fa = Foo_abstract(1.0)
    fc = Foo_concrete(1.0)

Out[34]: Foo_concrete{Float64}(1.0)

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

In [35]: typeof(fc)

Out[35]: 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

In [36]: function f(foo)
    
    for i = 1:1_000_000
        tmp = i + foo.a
    
    end
    
end

Out[36]: f (generic function with 2 methods)
Let's try timing our code, starting with the generic case:

In [37]: @btime f($fg)

    39.721 ms (1999489 allocations: 30.51 MiB)

The timing is not very impressive.

Here's the nasty looking machine code

In [38]: @code_native f(fg)
The abstract case is similar

In [39]: @btime f($fa)

39.609 ms (1999489 allocations: 30.51 MiB)

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

In [40]: @btime f($fc)

1.599 ns (0 allocations: 0 bytes)

Some of this time is JIT compilation, and one more execution gets us down to.
Here’s the corresponding machine code

In [41]: @code_native f(fc)

```
.text
; @ In[36]:2 within `f'
retq
  nopw %cs:(%rax,%rax)
  nopl (%rax,%rax)
; 
```

Much nicer…
14.6.4 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

```julia
In [42]: function sum_float_array(x::AbstractVector{<:Number})
    sum = 0.0
    for i in eachindex(x)
        sum += x[i]
    end
    return sum
end
```

```
Out[42]: sum_float_array (generic function with 1 method)
```

Calls to this function run very quickly

```julia
In [43]: x = range(0, 1, length = Int(1e6))
x = collect(x)
typeof(x)
```

```
Out[43]: Array{Float64,1}
```

```julia
In [44]: @btime sum_float_array($x)
```

```
1.251 ms (0 allocations: 0 bytes)
```

```
Out[44]: 499999.999999796
```

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

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

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

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

**Type Inferences**

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

```julia
In [45]: function sum_array(x)
    sum = 0.0
    for i in eachindex(x)
        sum += x[i]
    end
    return sum
end
```

```julia
```
```
Out[45]: sum_array (generic function with 1 method)

When we run it with the same array of floating point numbers it executes at a similar speed as the function with type information.

In [46]: @btime sum_array($x)

    1.251 ms (0 allocations: 0 bytes)

Out[46]: 499999.9999999796

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 we’re calling the function on a vector of floats, we get a compiled version of the function with essentially the same internal representation as sum_float_array().

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

In [47]: x = Any[ 1/i for i :: 1:1e6 ];

In [48]: eltype(x)

Out[48]: Any

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

In [49]: @btime sum_array($x)

    27.232 ms (1000000 allocations: 15.26 MiB)

Out[49]: 14.392726722864989

14.7 Further Comments

Here are some final comments on performance.
14.7. FURTHER COMMENTS

14.7.1 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 it’s 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 loose 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 it’s liable to get tripped up

14.7.2 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.

14.7.3 Further Reading

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

Tools and Techniques
Chapter 15

Linear Algebra

15.1 Contents

- Overview 15.2
- Vectors 15.3
- Matrices 15.4
- Solving Systems of Equations 15.5
- Eigenvalues and Eigenvectors 15.6
- Further Topics 15.7
- Exercises 15.8
- Solutions 15.9

15.2 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,

\[
\begin{align*}
y_1 &= a_{11}x_1 + a_{12}x_2 + \cdots + a_{1k}x_k \\
&\vdotswithin{=} \\
y_n &= a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nk}x_k
\end{align*}
\]

(1)

The objective here is to solve for the “unknowns” \(x_1, \ldots, x_k\) given \(a_{11}, \ldots, a_{nk}\) and \(y_1, \ldots, y_n\).

When considering such problems, it is essential that we first consider at least some of the following questions:

- Does a solution actually exist?
- Are there in fact many solutions, and if so how should we interpret them?
- If no solution exists, is there a best “approximate” solution?
If a solution exists, how should we compute it?

These are the kinds of topics addressed by linear algebra.

In this lecture we will cover the basics of linear and matrix algebra, treating both theory and computation.

We admit some overlap with this lecture, where operations on 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.

## 15.3 Vectors

A vector is an element of a vector space.

Vectors can be added together and scaled (multiplied) by scalars.

Vectors can be written as \( x = [x_1, \ldots, x_n] \).

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.

### 15.3.1 Setup

```
In [1]: using InstantiateFromURL
   # optionally add arguments to force installation: instantiate = true,
   # precompile = true
   github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

In [2]: using LinearAlgebra, Statistics, Plots
   gr(fmt=:png);

In [3]: x_vals = [0 0 0 ; 2 -3 -4]
y_vals = [0 0 0 ; 4 3 -3.5]

   plot(x_vals, y_vals, arrow = true, color = :blue,
   legend = :none, xlims = (-5, 5), ylims = (-5, 5),
   annotations = [(2.2, 4.4, "[2, 4]"),
                   (-3.3, 3.3, "[-3, 3]"),
                   (-4.4, -3.85, "[-4, -3.5]"),
                   (-5:1:5, yticks = -5:1:5,
                    framestyle = :origin)
   ```
15.3.2 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.

In [5]: \( x = \text{ones}(3) \)

Out[5]: 3-element Array{Float64,1}:
        1.0
        1.0
        1.0

In [6]: \( y = [2, 4, 6] \)

Out[6]: 3-element Array{Int64,1}:
        2
        4
        6

In [7]: \( x + y \)

Out[7]: 3-element Array{Float64,1}:
        3.0
        5.0
        7.0
15.3. VECTORS

In [8]: 4x  # equivalent to 4 * x and 4 .* x

Out[8]: 3-element Array{Float64,1}:
4.0
4.0
4.0

15.3.3 Inner Product and Norm

The inner product of vectors \(x, y \in \mathbb{R}^n\) is defined as

\[
x'y := \sum_{i=1}^{n} x_i y_i
\]

Two vectors are called orthogonal if their inner product is zero.

The norm of a vector \(x\) represents its “length” (i.e., its distance from the zero vector) and is defined as

\[
\|x\| := \sqrt{x'x} := \left(\sum_{i=1}^{n} x_i^2\right)^{1/2}
\]

The expression \(\|x - y\|\) is thought of as the distance between \(x\) and \(y\).

Continuing on from the previous example, the inner product and norm can be computed as follows

In [9]: using LinearAlgebra

In [10]: dot(x, y)  # Inner product of x and y

Out[10]: 12.0

In [11]: sum(prod, zip(x, y))  # Gives the same result

Out[11]: 12.0

In [12]: norm(x)  # Norm of x

Out[12]: 1.7320508075688772

In [13]: sqrt(sum(abs2, x))  # Gives the same result

Out[13]: 1.7320508075688772
15.3.4 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

\[
y = \beta_1 a_1 + \cdots + \beta_k a_k
\]

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.

In [14]:  

```python
# fixed linear function, to generate a plane
f(x, y) = 0.2x + 0.1y

# lines to vectors
x_vec = [0 0; 3 3]
y_vec = [0 0; 4 -4]
z_vec = [0 0; f(3, 4) f(3, -4)]

# draw the plane
n = 20
grid = range(-5, 5, length = n)
z2 = [ f(grid[row], grid[col]) for row in 1:n, col in 1:n ]
wireframe(grid, grid, z2, fill = :blues, gridalpha =1 )
plot!(x_vec, y_vec, z_vec, color = [:blue :green], linewidth = 3, labels
```

Out[14]:
Examples

If $A$ contains only one vector $a_1 \in \mathbb{R}^2$, then its span is just the scalar multiples of $a_1$, which is the unique line passing through both $a_1$ and the origin.

If $A = \{e_1, e_2, e_3\}$ consists of the canonical basis vectors of $\mathbb{R}^3$, that is

$$
e_1 := \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad e_2 := \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad e_3 := \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

then the span of $A$ is all of $\mathbb{R}^3$, because, for any $x = (x_1, x_2, x_3) \in \mathbb{R}^3$, we can write

$$x = x_1e_1 + x_2e_2 + x_3e_3$$

Now consider $A_0 = \{e_1, e_2, e_1 + e_2\}$.

If $y = (y_1, y_2, y_3)$ is any linear combination of these vectors, then $y_3 = 0$ (check it).

Hence $A_0$ fails to span all of $\mathbb{R}^3$.

15.3.5 Linear Independence

As we'll see, it’s often desirable to find families of vectors with relatively large span, so that many vectors can be described by linear operators on a few vectors.

The condition we need for a set of vectors to have a large span is what’s called linear independence.

In particular, a collection of vectors $A := \{a_1, \ldots, a_k\}$ in $\mathbb{R}^n$ is said to be

- linearly dependent if some strict subset of $A$ has the same span as $A$
• *linearly independent* if it is not linearly dependent

Put differently, a set of vectors is linearly independent if no vector is redundant to the span, and linearly dependent otherwise.

To illustrate the idea, recall the figure that showed the span of vectors \( \{a_1, a_2\} \) in \( \mathbb{R}^3 \) as a plane through the origin.

If we take a third vector \( a_3 \) and form the set \( \{a_1, a_2, a_3\} \), this set will be

- linearly dependent if \( a_3 \) lies in the plane
- linearly independent otherwise

As another illustration of the concept, since \( \mathbb{R}^n \) can be spanned by \( n \) vectors (see the discussion of canonical basis vectors above), any collection of \( m > n \) vectors in \( \mathbb{R}^n \) must be linearly dependent.

The following statements are equivalent to linear independence of \( A := \{a_1, \ldots, a_k\} \subset \mathbb{R}^n \).

1. No vector in \( A \) can be formed as a linear combination of the other elements.
2. If \( \beta_1 a_1 + \cdots + \beta_k a_k = 0 \) for scalars \( \beta_1, \ldots, \beta_k \), then \( \beta_1 = \cdots = \beta_k = 0 \).

(The zero in the first expression is the origin of \( \mathbb{R}^n \))

15.3.6 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 \( i \).

15.4 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 \(I\).

### 15.4.1 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.
\[ A x = \begin{bmatrix} a_{11} & \cdots & a_{1k} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nk} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_k \end{bmatrix} := \begin{bmatrix} a_{11}x_1 + \cdots + a_{1k}x_k \\ \vdots \\ a_{n1}x_1 + \cdots + a_{nk}x_k \end{bmatrix} \] (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 \( IA = A \).

### 15.4.2 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

In [15]: \( \text{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \)

Out[15]: \( 2\times2 \text{ Array\{Int64,2\}:} \)
\[ \begin{array}{ll} 1 & 2 \\ 3 & 4 \end{array} \]

In [16]: \( \text{typeof(A)} \)

Out[16]: \( \text{Array\{Int64,2\}} \)

In [17]: \( \text{size(A)} \)

Out[17]: \( (2, 2) \)

The \textbf{size} function returns a tuple giving the number of rows and columns.

To get the transpose of \( A \), use \textbf{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

In [18]: \( \text{A} = \text{ones}(3, 3) \)

Out[18]: \( 3\times3 \text{ Array\{Float64,2\}:} \)
\[ \begin{array}{ccc} 1.0 & 1.0 & 1.0 \\ 1.0 & 1.0 & 1.0 \\ 1.0 & 1.0 & 1.0 \end{array} \]
To multiply matrices we use the * operator.
In particular, \( A \times B \) is matrix multiplication, whereas \( A \cdot B \) is element by element multiplication.

### 15.4.3 Matrices as Maps

Each \( n \times k \) matrix \( A \) can be identified with a function \( f(x) = Ax \) that maps \( x \in \mathbb{R}^k \) into \( y = Ax \in \mathbb{R}^n \).

These kinds of functions have a special property: they are linear.
A function \( f: \mathbb{R}^k \rightarrow \mathbb{R}^n \) is called linear if, for all \( x, y \in \mathbb{R}^k \) and all scalars \( \alpha, \beta \), we have

\[
    f(\alpha x + \beta y) = \alpha f(x) + \beta f(y)
\]

You can check that this holds for the function \( f(x) = Ax + b \) when \( b \) is the zero vector, and fails when \( b \) is nonzero.

In fact, it’s known that \( f \) is linear if and only if there exists a matrix \( A \) such that \( f(x) = Ax \) for all \( x \).

### 15.5 Solving Systems of Equations

Recall again the system of equations (1).
If we compare (1) and (2), we see that (1) can now be written more conveniently as

\[
    y = Ax
\]

The problem we face is to determine a vector \( x \in \mathbb{R}^k \) that solves (3), taking \( y \) and \( A \) as given.
This is a special case of a more general problem: Find an \( x \) such that \( y = f(x) \).
Given an arbitrary function \( f \) and a \( y \), is there always an \( x \) such that \( y = f(x) \)?
If so, is it always unique?
The answer to both these questions is negative, as the next figure shows.
In the first plot there are multiple solutions, as the function is not one-to-one, while in the second there are no solutions, since $y$ lies outside the range of $f$.

Can we impose conditions on $A$ in (3) that rule out these problems?

In this context, the most important thing to recognize about the expression $Ax$ is that it corresponds to a linear combination of the columns of $A$.

In particular, if $a_1, \ldots, a_k$ are the columns of $A$, then

$$Ax = x_1a_1 + \cdots + x_ka_k$$

Hence the range of $f(x) = Ax$ is exactly the span of the columns of $A$.

We want the range to be large, so that it contains arbitrary $y$. 
15.5. SOLVING SYSTEMS OF EQUATIONS

As you might recall, the condition that we want for the span to be large is linear independence.

A happy fact is that linear independence of the columns of $A$ also gives us uniqueness.

Indeed, it follows from our earlier discussion that if $\{a_1, \ldots, a_k\}$ are linearly independent and $y = Ax = x_1a_1 + \cdots + x_ka_k$, then no $z \neq x$ satisfies $y = Az$.

15.5.1 The $n \times n$ Case

Let’s discuss some more details, starting with the case where $A$ is $n \times n$.

This is the familiar case where the number of unknowns equals the number of equations.

For arbitrary $y \in \mathbb{R}^n$, we hope to find a unique $x \in \mathbb{R}^n$ such that $y = Ax$.

In view of the observations immediately above, if the columns of $A$ are linearly independent, then their span, and hence the range of $f(x) = Ax$, is all of $\mathbb{R}^n$.

Hence there always exists an $x$ such that $y = Ax$.

Moreover, the solution is unique.

In particular, the following are equivalent

1. The columns of $A$ are linearly independent
2. For any $y \in \mathbb{R}^n$, the equation $y = Ax$ has a unique solution

The property of having linearly independent columns is sometimes expressed as having full column rank.

Inverse Matrices

Can we give some sort of expression for the solution?

If $y$ and $A$ are scalar with $A \neq 0$, then the solution is $x = A^{-1}y$.

A similar expression is available in the matrix case.

In particular, if square matrix $A$ has full column rank, then it possesses a multiplicative inverse matrix $A^{-1}$, with the property that $AA^{-1} = A^{-1}A = I$.

As a consequence, if we pre-multiply both sides of $y = Ax$ by $A^{-1}$, we get $x = A^{-1}y$.

This is the solution that we’re looking for.

Determinants

Another quick comment about square matrices is that to every such matrix we assign a unique number called the determinant of the matrix — you can find the expression for it here.

If the determinant of $A$ is not zero, then we say that $A$ is nonsingular.

Perhaps the most important fact about determinants is that $A$ is nonsingular if and only if $A$ is of full column rank.
This gives us a useful one-number summary of whether or not a square matrix can be inverted.

15.5.2 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

15.5.3 More Columns than Rows

This is the $n \times k$ case with $n < k$, so there are fewer equations than unknowns.

In this case there are either no solutions or infinitely many — in other words, uniqueness never holds.

For example, consider the case where $k = 3$ and $n = 2$.

Thus, the columns of $A$ consists of 3 vectors in $\mathbb{R}^2$.

This set can never be linearly independent, since it is possible to find two vectors that span $\mathbb{R}^2$.

(For example, use the canonical basis vectors)

It follows that one column is a linear combination of the other two.

For example, let’s say that $a_1 = \alpha a_2 + \beta a_3$.

Then if $y = Ax = x_1 a_1 + x_2 a_2 + x_3 a_3$, we can also write
\[ y = x_1(\alpha a_2 + \beta a_3) + x_2a_2 + x_3a_3 = (x_1\alpha + x_2)a_2 + (x_1\beta + x_3)a_3 \]

In other words, uniqueness fails.

15.5.4 Linear Equations with Julia

Here’s an illustration of how to solve linear equations with Julia’s built-in linear algebra facilities

In [22]: `A = [1.0 2.0; 3.0 4.0];`

In [23]: `y = ones(2, 1); # A column vector`

In [24]: `det(A)`

Out[24]: `-2.0`

In [25]: `A_inv = inv(A)`

Out[25]: `2×2 Array{Float64,2}:
       -2.0  1.0
       1.5  -0.5`

In [26]: `x = A_inv * y # solution`

Out[26]: `2×1 Array{Float64,2}:
      -0.9999999999999998
      0.9999999999999999`

In [27]: `A * x # should equal y (a vector of ones)`

Out[27]: `2×1 Array{Float64,2}:
     1.0000000000000004
     1.0000000000000004`

In [28]: `A \ y # produces the same solution`

Out[28]: `2×1 Array{Float64,2}:
     -1.0
     1.0`

Observe how we can solve for \(x = A^{-1}y\) by either via \(\text{inv}(A) \ast y\), or using \(A \backslash y\).

The latter method is preferred because it automatically selects the best algorithm for the problem based on the types of \(A\) and \(y\).

If \(A\) is not square then \(A \backslash y\) returns the least squares solution \(\hat{x} = (A'A)^{-1}A'y\).
15.6 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.

In [29]:
A = [1 2
     2 1]
evals, evecs = eigen(A)
a1, a2 = evals
eig_1 = [0 0; evecs[:,1]']
eig_2 = [0 0; evecs[:,2]']
x = range(-5, 5, length = 10)
y = -x

plot(eig_1[:, 2], a1 * eig_2[:, 2], arrow = true, color = :red,
     legend = :none, xticks = (-3, 3), yticks = (-3, 3), xticks = -3:3,
     yticks = -3:3,
     framestyle = :origin)
plot!(a2 * eig_1[:, 2], a2 * eig_2, arrow = true, color = :red)
plot!(eig_1, eig_2, arrow = true, color = :blue)
plot!(x, y, color = :blue, lw = 0.4, alpha = 0.6)
plot!(x, x, color = :blue, lw = 0.4, alpha = 0.6)
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

\begin{verbatim}
In [30]: A = [1.0 2.0; 2.0 1.0];
In [31]: evals, evecs = eigen(A);
\end{verbatim}
In [32]: evals

Out[32]: 2-element Array{Float64,1}:
  -1.0
  3.0

In [33]: evecs

Out[33]: 2×2 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.

15.6.1 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 `eigen(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.

15.7 Further Topics

We round out our discussion by briefly mentioning several other important topics.

15.7.1 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 Section ??.
15.7. FURTHER TOPICS

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

$$(I - A)^{-1} = \sum_{k=0}^{\infty} A^k$$

(4)

Spectral Radius

A result known as Gelfand’s formula tells us that, for any square matrix $A$,

$$\rho(A) = \lim_{k \to \infty} \|A^k\|^{1/k}$$

Here $\rho(A)$ is the spectral radius, defined as $\max_i |\lambda_i|$, where $\{\lambda_i\}_i$ is the set of eigenvalues of $A$.

As a consequence of Gelfand’s formula, if all eigenvalues are strictly less than one in modulus, there exists a $k$ with $\|A^k\| < 1$.

In which case (4) is valid.

15.7.2 Positive Definite Matrices

Let $A$ be a symmetric $n \times n$ matrix.

We say that $A$ is

1. **positive definite** if $x'Ax > 0$ for every $x \in \mathbb{R}^n \setminus \{0\}$

2. **positive semi-definite** or **nonnegative definite** if $x'Ax \geq 0$ for every $x \in \mathbb{R}^n$

Analogous definitions exist for negative definite and negative semi-definite matrices.

It is notable that if $A$ is positive definite, then all of its eigenvalues are strictly positive, and hence $A$ is invertible (with positive definite inverse).

15.7.3 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. $\frac{\partial y'z}{\partial x} = a$
2. \( \frac{\partial A x}{\partial x} = A' \)
3. \( \frac{\partial x' A x}{\partial x} = (A + A')x \)
4. \( \frac{\partial y' B z}{\partial y} = B z \)
5. \( \frac{\partial y' B z}{\partial B} = y z' \)

Exercise 1 below asks you to apply these formulas.

15.7.4 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 [57].

15.8 Exercises

15.8.1 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 
   \[ v(x) = -x'\hat{P}x \] 
   where 
   \[ \hat{P} = A'PA - A'PB(Q + B'PB)^{-1}B'PA \]

As we will see, in economic contexts Lagrange multipliers often are shadow prices

**Note**

If we don’t care about the Lagrange multipliers, we can substitute the constraint into the objective function, and then just maximize 
\[ -(Ax+Bu)'P(Ax+Bu) - u'Qu \]

with respect to \( u \). You can verify that this leads to the same maximizer.

## 15.9 Solutions

Thanks to Willem Hekman and Guanlong Ren for providing this solution.

### 15.9.1 Exercise 1

We have an optimization problem:

\[ v(x) = \max_{y,u} \{-y'P'y - u'Qu\} \]

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:

\[ L = -y'P'y - u'Qu + \lambda'[Ax + Bu - y] \]

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 $u = -(Q + B'PB)^{-1}B'PAx$, then

$$v(x) = -(Ax + Bu)'P(Ax + Bu) - u'Qu \text{ with } u = -(Q + B'PB)^{-1}B'PAx$$

To evaluate the function

$$v(x) = -(Ax + Bu)'P(Ax + Bu) - u'Qu$$

$$= -(x'A' + u'B')P(Ax + Bu) - u'Qu$$

$$= -x'A'PAx - u'B'PAx - x'A'PBu - u'B'PBu - u'Qu$$

$$= -x'A'PAx - 2u'B'PAx - u'(Q + B'PB)u$$

For simplicity, denote by $S := (Q + B'PB)^{-1}B'PA$, then $u = -Sx$.

Regarding the second term $-2u'B'PAx$,

$$-2u'B'PAx = -2x'S'B'PAx$$

$$= 2x'A'PB(Q + B'PB)^{-1}B'PAx$$
Notice that the term \((Q + B'PB)^{-1}\) is symmetric as both \(P\) and \(Q\) are symmetric.

Regarding the third term \(-u'(Q + B'PB)u\),

\[-u'(Q + B'PB)u = -x'\tilde{S}(Q + B'PB)x\]
\[= -x'A'PB(Q + B'PB)^{-1}B'PAx\]

Hence, the summation of second and third terms is \(x'A'PB(Q + B'PB)^{-1}B'PAx\).

This implies that

\[v(x) = -x'A'PAx - 2u'B'PAx - u'(Q + B'PB)u\]
\[= -x'A'PAx + x'A'PB(Q + B'PB)^{-1}B'PAx\]
\[= -x'[A'PA - A'PB(Q + B'PB)^{-1}B'PA]x\]

Therefore, the solution to the optimization problem \(v(x) = -x'\tilde{P}x\) follows the above result by denoting \(\tilde{P} := A'PA - A'PB(Q + B'PB)^{-1}B'PA\).

**Footnotes**

[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.
Chapter 16

Orthogonal Projections and Their Applications

16.1 Contents

- Overview 16.2
- Key Definitions 16.3
- The Orthogonal Projection Theorem 16.4
- Orthonormal Basis 16.5
- Projection Using Matrix Algebra 16.6
- Least Squares Regression 16.7
- Orthogonalization and Decomposition 16.8
- Exercises 16.9
- Solutions 16.10

16.2 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

16.2.1 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, [90].

For an advanced treatment of projection in the context of least squares prediction, see this book chapter.

### 16.3 Key Definitions

Assume $x, z \in \mathbb{R}^n$.

Define $\langle x, z \rangle = \sum_i x_i z_i$.

Recall $\|x\|^2 = \langle x, x \rangle$.

The law of cosines states that $\langle x, z \rangle = \|x\|\|z\| \cos(\theta)$ where $\theta$ is the angle between the vectors $x$ and $z$.

When $\langle x, z \rangle = 0$, then $\cos(\theta) = 0$ and $x$ and $z$ are said to be orthogonal and we write $x \perp z$.

For a linear subspace $S \subset \mathbb{R}^n$, we call $x \in \mathbb{R}^n$ orthogonal to $S$ if $x \perp z$ for all $z \in S$, and write $x \perp S$. 
The orthogonal complement of linear subspace $S \subset \mathbb{R}^n$ 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

\[ \|x_1 + x_2\|^2 = \langle x_1 + x_2, x_1 + x_2 \rangle = \langle x_1, x_1 \rangle + 2\langle x_2, x_1 \rangle + \langle x_2, x_2 \rangle = \|x_1\|^2 + \|x_2\|^2 \]

### 16.3.1 Linear Independence vs Orthogonality

If \( X \subset \mathbb{R}^n \) is an orthogonal set and \( 0 \notin X \), then \( X \) is linearly independent.

Proving this is a nice exercise.

While the converse is not true, a kind of partial converse holds, as we’ll see below.

### 16.4 The Orthogonal Projection Theorem

What vector within a linear subspace of \( \mathbb{R}^n \) best approximates a given vector in \( \mathbb{R}^n \)?

The next theorem provides answers this question.

**Theorem (OPT)** Given \( y \in \mathbb{R}^n \) and linear subspace \( S \subset \mathbb{R}^n \), there exists a unique solution to the minimization problem

\[
\hat{y} := \text{arg min}_{z \in S} \| y - z \|
\]

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...
16.4. PROOF OF SUFFICIENCY

We’ll omit the full proof.
But we will prove sufficiency of the asserted conditions.
To this end, let \( y \in \mathbb{R}^n \) and let \( S \) be a linear subspace of \( \mathbb{R}^n \).
Let \( \hat{y} \) be a vector in \( \mathbb{R}^n \) such that \( \hat{y} \in S \) and \( y - \hat{y} \perp S \).
Let \( z \) be any other point in \( S \) and use the fact that \( S \) is a linear subspace to deduce
\[
\|y - z\|^2 = \|(y - \hat{y}) + (\hat{y} - z)\|^2 = \|y - \hat{y}\|^2 + \|\hat{y} - z\|^2
\]
Hence \( \|y - z\| \geq \|y - \hat{y}\| \), which completes the proof.

16.4.2 ORTHOGONAL PROJECTION AS A MAPPING

For a linear space \( Y \) and a fixed linear subspace \( S \), we have a functional relationship
\[
y \in Y \mapsto \text{its orthogonal projection } \hat{y} \in S
\]
By the OPT, this is a well-defined mapping or operator from \( \mathbb{R}^n \) to \( \mathbb{R}^n \).
In what follows we denote this operator by a matrix \( P \)
- \( Py \) represents the projection \( \hat{y} \).
- This is sometimes expressed as \( \hat{E}_S y = Py \), where \( \hat{E} \) denotes a wide-sense expectations operator and the subscript \( S \) indicates that we are projecting \( y \) onto the linear subspace \( S \).
The operator $P$ is called the **orthogonal projection mapping onto** $S$

It is immediate from the OPT that for any $y \in \mathbb{R}^n$

1. $Py \in S$ and
2. $y - Py \perp S$

From this we can deduce additional useful properties, such as

1. $\|y\|^2 = \|Py\|^2 + \|y - Py\|^2$ and
2. $\|Py\| \leq \|y\|$

For example, to prove 1, observe that $y = Py + y - Py$ and apply the Pythagorean law.

**Orthogonal Complement**

Let $S \subset \mathbb{R}^n$.

The **orthogonal complement** of $S$ is the linear subspace $S^\perp$ that satisfies $x_1 \perp x_2$ for every $x_1 \in S$ and $x_2 \in S^\perp$.

Let $Y$ be a linear space with linear subspace $S$ and its orthogonal complement $S^\perp$.

We write

$$Y = S \oplus S^\perp$$

to indicate that for every $y \in Y$ there is unique $x_1 \in S$ and a unique $x_2 \in S^\perp$ such that $y = x_1 + x_2$. 
Moreover, \( x_1 = \hat{E}_S y \) and \( x_2 = y - \hat{E}_S y \).

This amounts to another version of the OPT:

**Theorem.** If \( S \) is a linear subspace of \( \mathbb{R}^n \), \( \hat{E}_S y = Py \) and \( \hat{E}_S \perp y = My \), then

\[
Py \perp My \quad \text{and} \quad y = Py + My \quad \text{for all} \ y \in \mathbb{R}^n
\]

The next figure illustrates

\[ S \perp \]

\[ My \]

\[ Py \]

\[ S \]

16.5 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

\[
x = \sum_{i=1}^{k} \langle x, u_i \rangle u_i \quad \text{for all} \ x \in S
\]

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
\[ x = \sum_{j=1}^{k} \alpha_j u_j \]  

(1)

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 (1) verifies the claim.

### 16.5.1 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)

**Proof:** Fix \( y \in \mathbb{R}^n \) and let \( P y \) be defined as in (2).

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

\[ \langle y - \sum_{i=1}^{k} \langle y, u_i \rangle u_i, u_j \rangle = \langle y, u_j \rangle - \sum_{i=1}^{k} \langle y, u_i \rangle \langle u_i, u_j \rangle = 0 \]

### 16.6 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
16.6. PROJECTION USING MATRIX ALGEBRA

1. \( Py \in S \), and
2. \( y - Py \perp S \)

Claim 1 is true because

\[
Py = X(X'X)^{-1}X'y = Xa \quad \text{when} \quad a := (X'X)^{-1}X'y
\]

An expression of the form \( Xa \) is precisely a linear combination of the columns of \( X \), and hence an element of \( S \).

Claim 2 is equivalent to the statement

\[
y - X(X'X)^{-1}X'y \perp Xb \quad \text{for all} \quad b \in \mathbb{R}^K
\]

This is true: If \( b \in \mathbb{R}^K \), then

\[
(Xb)'[y - X(X'X)^{-1}X'y] = b'[X'y - X'y] = 0
\]

The proof is now complete.

16.6.1 Starting with \( X \)

It is common in applications to start with \( n \times k \) matrix \( X \) with linearly independent columns and let

\[
S := \text{span} X := \text{span}\{\col 1 X, \ldots, \col k X\}
\]

Then the columns of \( X \) form a basis of \( S \).

From the preceding theorem, \( P = X(X'X)^{-1}X'y \) projects \( y \) onto \( S \).

In this context, \( P \) is often called the projection matrix.

- The matrix \( M = I - P \) satisfies \( My = \hat{E}_S y \) and is sometimes called the annihilator matrix.

16.6.2 The Orthonormal Case

Suppose that \( U \) is \( n \times k \) with orthonormal columns.

Let \( u_i := \col U_i \) for each \( i \), let \( S := \text{span} U \) and let \( y \in \mathbb{R}^n \).

We know that the projection of \( y \) onto \( S \) is

\[
Py = U(U'U)^{-1}U'y
\]

Since \( U \) has orthonormal columns, we have \( U'U = I \).

Hence
\[ Py = UU'y = \sum_{i=1}^{k} (u_i, y)u_i \]

We have recovered our earlier result about projecting onto the span of an orthonormal basis.

### 16.6.3 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.

### 16.7 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.
16.7. LEAST SQUARES REGRESSION

16.7.1 Squared risk measures

Given pairs \((x, y) \in \mathbb{R}^K \times \mathbb{R}\), consider choosing \(f : \mathbb{R}^K \to \mathbb{R}\) to minimize the risk:

\[
R(f) := \mathbb{E} [(y - f(x))^2]
\]

If probabilities and hence \(\mathbb{E}\) are unknown, we cannot solve this problem directly. However, if a sample is available, we can estimate the risk with the empirical risk:

\[
\min_{f \in \mathcal{F}} \frac{1}{N} \sum_{n=1}^{N} (y_n - f(x_n))^2
\]

Minimizing this expression is called empirical risk minimization.

The set \(\mathcal{F}\) is sometimes called the hypothesis space.

The theory of statistical learning tells us that to prevent overfitting we should take the set \(\mathcal{F}\) to be relatively simple.

If we let \(\mathcal{F}\) be the class of linear functions \(1/N\), the problem is:

\[
\min_{b \in \mathbb{R}^K} \sum_{n=1}^{N} (y_n - b'x_n)^2
\]

This is the sample linear least squares problem.

16.7.2 Solution

Define the matrices

\[
y := \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix}, \quad x_n := \begin{pmatrix} x_{n1} \\ x_{n2} \\ \vdots \\ x_{nK} \end{pmatrix} = n-th obs on all regressors
\]

and

\[
X := \begin{pmatrix} x_1' \\ x_2' \\ \vdots \\ x_N' \end{pmatrix} := \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1K} \\ x_{21} & x_{22} & \cdots & x_{2K} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N1} & x_{N2} & \cdots & x_{NK} \end{pmatrix}
\]

We assume throughout that \(N > K\) and \(X\) is full column rank.

If you work through the algebra, you will be able to verify that \(\|y - Xb\|^2 = \sum_{n=1}^{N} (y_n - b'x_n)^2\).

Since monotone transforms don’t affect minimizers, we have

\[
\arg \min_{b \in \mathbb{R}^K} \sum_{n=1}^{N} (y_n - b'x_n)^2 = \arg \min_{b \in \mathbb{R}^K} \|y - Xb\|
\]
By our results about overdetermined linear systems of equations, the solution is

\[
\hat{\beta} := (X'X)^{-1}X'y
\]

Let \( P \) and \( M \) be the projection and annihilator associated with \( X \):

\[
P := X(X'X)^{-1}X' \quad \text{and} \quad M := I - P
\]

The vector of fitted values is

\[
\hat{y} := X\hat{\beta} = Py
\]

The vector of residuals is

\[
\hat{u} := y - \hat{y} = y - Py = My
\]

Here are some more standard definitions:

- The total sum of squares is \( \|y\|^2 \).
- The sum of squared residuals is \( \|\hat{u}\|^2 \).
- The explained sum of squares is \( \|\hat{y}\|^2 \).

\[
\text{TSS} = \text{ESS} + \text{SSR}
\]

We can prove this easily using the OPT.

From the OPT we have \( y = \hat{y} + \hat{u} \) and \( \hat{u} \perp \hat{y} \).

Applying the Pythagorean law completes the proof.

### 16.8 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.

#### 16.8.1 Gram-Schmidt Orthogonalization

**Theorem** For each linearly independent set \( \{x_1, \ldots, x_k\} \subset \mathbb{R}^n \), there exists an orthonormal set \( \{u_1, \ldots, u_k\} \) with

\[
\text{span}\{x_1, \ldots, x_i\} = \text{span}\{u_1, \ldots, u_i\} \quad \text{for} \quad i = 1, \ldots, k
\]

The Gram-Schmidt orthogonalization procedure constructs an orthogonal set \( \{u_1, u_2, \ldots, u_n\} \).

One description of this procedure is as follows:
• For $i = 1, \ldots, k$, form $S_i := \text{span}\{x_1, \ldots, x_i\}$ and $S_i^\perp$
• Set $v_1 = x_1$
• For $i \geq 2$ set $v_i := \hat{E}_{S_i^\perp} x_i$ 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.

### 16.8.2 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 $X = QR$.

### 16.8.3 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

$$\hat{\beta} = (R'Q'QR)^{-1}R'Q'y$$

$$= (R'R)^{-1}R'Q'y$$

$$= R^{-1}(R')^{-1}R'Q'y = R^{-1}Q'y$$

Numerical routines would in this case use the alternative form $R\hat{\beta} = Q'y$ and back substitution.
16.9 Exercises

16.9.1 Exercise 1

Show that, for any linear subspace \( S \subset \mathbb{R}^n \), \( S \cap S^\perp = \{0\} \).

16.9.2 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?

16.9.3 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

\[
X := \begin{pmatrix} 1 & 0 \\ 0 & -6 \\ 2 & 2 \end{pmatrix}
\]

16.10 Solutions

16.10.1 Exercise 1

If \( x \in S \) and \( x \in S^\perp \), then we have in particular that \( \langle x, x \rangle = 0 \). But then \( x = 0 \).

16.10.2 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).

16.10.3 Exercise 3

Here’s a function that computes the orthonormal vectors using the GS algorithm given in the lecture.
16.10.4 Setup

In [1]: using InstantiateFromURL
   # optionally add arguments to force installation: instantiate = true,
   # precompile = true
   github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

In [2]: using LinearAlgebra, Statistics

In [3]: function gram_schmidt(X)
   
   U = similar(X, Float64) # for robustness
   
   function normalized_orthogonal_projection(b, Z)
     # project onto the orthogonal complement of the col span of Z
     orthogonal = I - Z * inv(Z'Z) * Z'
     projection = orthogonal * b
     # normalize
     return projection / norm(projection)
   end
   
   for col in 1:size(U, 2)
     # set up
     b = X[:,col] # vector we're going to project
     Z = X[:,1:col - 1] # first i-1 columns of X
     U[:,col] = normalized_orthogonal_projection(b, Z)
   end
   
   return U
end

Out[3]: gram_schmidt (generic function with 1 method)

Here are the arrays we'll work with

In [4]: y = [1, 3, -3]
   X = [1 0; 0 -6; 2 2];

First let's do ordinary projection of y onto the basis spanned by the columns of X.

In [5]: Py1 = X * inv(X'X) * X' * y

Out[5]: 3-element Array{Float64,1}:
   -0.5652173913043479
   3.260869565217391
   -2.217391304347826

Now let's orthogonalize first, using Gram–Schmidt:

In [6]: U = gram_schmidt(X)

Out[6]: 3×2 Array{Float64,2}:
   0.447214  -0.131876
   0.0      -0.989071
   0.894427  0.065938
Now we can project using the orthonormal basis and see if we get the same thing:

\[
\begin{align*}
\text{In [7]: } & \quad \text{Py2} = U^* U' * y \\
\text{Out[7]: } & \quad 3\text{-element Array}\{\text{Float64,1}\}:
-0.5652173913043477 \\
& \quad \quad \quad 3.260869565217391 \\
& \quad \quad \quad -2.2173913043478257
\end{align*}
\]

The result is the same. To complete the exercise, we get an orthonormal basis by QR decomposition and project once more.

\[
\begin{align*}
\text{In [8]: } & \quad Q, R = \text{qr}(X) \\
& \quad Q = \text{Matrix}(Q) \\
\text{Out[8]: } & \quad 3\times2\text{ Array}\{\text{Float64,2}\}:
-0.447214 & -0.131876 \\
0.0 & -0.989071 \\
-0.894427 & 0.065938
\end{align*}
\]

\[
\begin{align*}
\text{In [9]: } & \quad \text{Py3} = Q * Q' * y \\
\text{Out[9]: } & \quad 3\text{-element Array}\{\text{Float64,1}\}:
-0.5652173913043474 \\
& \quad \quad \quad 3.2608695652173907 \\
& \quad \quad \quad -2.2173913043478253
\end{align*}
\]

Again, the result is the same.
Chapter 17

LLN and CLT

17.1 Contents

- Overview 17.2
- Relationships 17.3
- LLN 17.4
- CLT 17.5
- Exercises 17.6
- Solutions 17.7

17.2 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.

17.3 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.
17.4 LLN

We begin with the law of large numbers, which tells us when sample averages will converge to their population means.

17.4.1 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:

$$\mu := \mathbb{E}X = \int xF(dx)$$

In addition, let

$$\bar{X}_n := \frac{1}{n} \sum_{i=1}^{n} X_i$$

Kolmogorov’s strong law states that, if $\mathbb{E}|X|$ is finite, then

$$\Pr\left\{ \bar{X}_n \rightarrow \mu \text{ as } n \rightarrow \infty \right\} = 1 \quad (1)$$

What does this last expression mean?

Let’s think about it from a simulation perspective, imagining for a moment that our computer can generate perfect random samples (which of course it can’t).

Let’s also imagine that we can generate infinite sequences, so that the statement $\bar{X}_n \rightarrow \mu$ can be evaluated.

In this setting, (1) should be interpreted as meaning that the probability of the computer producing a sequence where $\bar{X}_n \rightarrow \mu$ fails to occur is zero.

17.4.2 Proof

The proof of Kolmogorov’s strong law is nontrivial – see, for example, theorem 8.3.5 of [26].

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

$$\Pr\left\{ |\bar{X}_n - \mu| \geq \epsilon \right\} \rightarrow 0 \text{ as } n \rightarrow \infty \quad (2)$$
(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

$$\Pr\{|\bar{X}_n - \mu| \geq \epsilon\} \leq \frac{\mathbb{E}[(\bar{X}_n - \mu)^2]}{\epsilon^2} \tag{3}$$

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 (3), we come to the estimate

$$\Pr\{|\bar{X}_n - \mu| \geq \epsilon\} \leq \frac{\sigma^2}{n\epsilon^2} \tag{4}$$

The claim in (2) 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.

### 17.4.3 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.

17.4.4 Setup

In [1]: using InstantiateFromURL
   # optionally add arguments to force installation: instantiate = true,
   # precompile = true
   github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

In [2]: using LinearAlgebra, Statistics
   using Plots, Distributions, Random, Statistics
   gr(fmt = :png, size = (900, 500))

Out[2]: Plots.GRBackend()

In [3]: function ksl(distribution, n = 100)
   title = nameof(typeof(distribution))
   observations = rand(distribution, n)
   sample_means = cumsum(observations) ./ (1:n)
   μ = mean(distribution)
   plot(repeat((1:n)', 2),
       [zeros(1, n); observations'], label = "", color = :grey, alpha = 0.5),
   plot!(1:n, observations, color = :grey, markershape = :circle,
       alpha = 0.5, label = "", linewidth = 0)
   if !isnan(μ)
       hline!(μ, color = :black, linewidth = 1.5, linestyle = :dash,
       grid = false,
       label = ["Mean"])
   end
   plot!(1:n, sample_means, linewidth = 3, alpha = 0.6, color = :green,
       label = "Sample mean")
   return plot!(title = title)
end

Out[3]: ksl (generic function with 2 methods)

In [4]: distributions = [TDist(10), Beta(2, 2), Gamma(5, 2), Poisson(4),
   LogNormal(0.5), Exponential(1)]

Out[4]: 6-element Array{Distribution{Univariate,S} where S<:ValueSupport,1}:
   TDist(Float64)(ν=10.0)
   Beta{Float64}(α=2.0, β=2.0)
   Gamma{Float64}(α=5.0, θ=2.0)
Here is an example for the standard normal distribution

In [5]: `ksl(Normal())`

Out[5]:

In [6]: `Random.seed!(0); # reproducible results`

In [7]: `plot(ksl(sample(distributions, 3, replace = false)), layout = (3, 1), legend = false)`

Out[7]:

- **Normal**
- **TDist**
- **Poisson**
- **Beta**
The three distributions are chosen at random from distributions.

### 17.4.5 Infinite Mean

What happens if the condition $\mathbb{E}|X| < \infty$ in the statement of the LLN is not satisfied? This might be the case if the underlying distribution is heavy tailed — the best known example is the Cauchy distribution, which has density

$$f(x) = \frac{1}{\pi(1 + x^2)} \quad (x \in \mathbb{R})$$

The next figure shows 100 independent draws from this distribution.

In [8]: Random.seed!(0); # reproducible results

In [9]: ksl(Cauchy())

Out[9]:

![Cauchy Distribution](image)

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.

In [10]: Random.seed!(0); # reproducible results

In [11]: function plot_means(n = 1000)
    sample_mean = cumsum(rand(Cauchy(), n)) ./ (1:n)
    plot(1:n, sample_mean, color = :red, alpha = 0.6, label = "Sample Mean", linewidth = 3)
    return hline!([0], color = :black, linestyle = :dash, label = "", grid = false)
Here we’ve increased \( n \) to 1000, but the sequence still shows no sign of converging. Will convergence become visible if we take \( n \) even larger? The answer is no.

To see this, recall that the characteristic function of the Cauchy distribution is

\[
\phi(t) = \mathbb{E}e^{itX} = \int e^{itx} f(x) dx = e^{-|t|} \tag{5}
\]

Using independence, the characteristic function of the sample mean becomes

\[
\mathbb{E}e^{it\bar{X}_n} = \mathbb{E}\left\{ e^{it\frac{1}{n} \sum_{j=1}^{n} X_j} \right\} \\
= \mathbb{E} \prod_{j=1}^{n} \exp \left\{ i \frac{t}{n} X_j \right\} \\
= \prod_{j=1}^{n} \mathbb{E} \exp \left\{ i \frac{t}{n} X_j \right\} = \left[ \phi(t/n) \right]^n
\]

In view of (5), this is just \( e^{-|t|} \).

Thus, in the case of the Cauchy distribution, the sample mean itself has the very same Cauchy distribution, regardless of \( n \).

In particular, the sequence \( \bar{X}_n \) does not converge to a point.
17.5 CLT

Next we turn to the central limit theorem, which tells us about the distribution of the deviation between sample averages and population means.

17.5.1 Statement of the Theorem

The central limit theorem is one of the most remarkable results in all of mathematics. In the classical IID setting, it tells us the following:

If the sequence $X_1, \ldots, X_n$ is IID, with common mean $\mu$ and common variance $\sigma^2 \in (0, \infty)$, then

$$\sqrt{n}(\bar{X}_n - \mu) \xrightarrow{d} N(0, \sigma^2) \quad \text{as} \quad n \to \infty$$

(6)

Here $\xrightarrow{d} N(0, \sigma^2)$ indicates convergence in distribution to a centered (i.e, zero mean) normal with standard deviation $\sigma$.

17.5.2 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 [26]).

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 $n = 1, 2, 4, 8$

In [12]: binomial_pdf(n) =
    bar(0:n, pdf.(Binomial(n), 0:n),
         xticks = 0:10, ylim = (0, 1), yticks = 0:0.1:1,
         label = "Binomial($n, 0.5$)", legend = :topleft)

Out[12]: binomial_pdf (generic function with 1 method)

In [13]: plot(binomial_pdf.((1,2,4,8)))...

Out[13]:

"Binomial($n, 0.5$)"
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.

### 17.5.3 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)

```plaintext
In [14]: using StatsPlots

function simulation1(distribution, n = 250, k = 10_000)
    σ = std(distribution)
    y = rand(distribution, n, k)
    y .= mean(distribution)
    y = mean(y, dims = 1)
    y = √n * vec(y)
    density(y, label = "Empirical Distribution")
    return plot!(Normal(0, σ), linestyle = :dash, color = :black,
                 label = "Normal(0.00, σ^2)")
end

Out[14]: simulation1 (generic function with 3 methods)

In [15]: simulation1(Exponential(0.5))

Out[15]:

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 $F$.

17.5.4 Simulation 2

Our next simulation is somewhat like the first, except that we aim to track the distribution of $Y_n := \sqrt{n}(\bar{X}_n - \mu)$ as $n$ increases.
In the simulation we’ll be working with random variables having $\mu = 0$.

Thus, when $n = 1$, we have $Y_1 = X_1$, so the first distribution is just the distribution of the underlying random variable.

For $n = 2$, the distribution of $Y_2$ is that of $(X_1 + X_2)/\sqrt{2}$, and so on.

What we expect is that, regardless of the distribution of the underlying random variable, the distribution of $Y_n$ will smooth out into a bell shaped curve.

The next figure shows this process for $X_i \sim f$, where $f$ was specified as the convex combination of three different beta densities.

(Taking a convex combination is an easy way to produce an irregular shape for $f$)

```python
In [16]: function simulation2(distribution = Beta(2, 2), n = 5, k = 10_000)
  y = rand(distribution, k, n)
  for col in 1:n
    y[:, col] += rand([-0.5, 0.6, -1.1], k)
  end
  y = (y .- mean(distribution))./std(distribution)
  y = cumsum(y, dims = 2)./sqrt.(1:5)' # return grid of data
end

Out[16]: simulation2 (generic function with 4 methods)

In [17]: ys = simulation2()
plots = [] # would preallocate in optimized code
for i in 1:size(ys, 2)
  p = density(ys[:, i], linealpha = i, title = "n = $i")
  push!(plots, p)
end
plot(plots..., legend = false)
```

Out[17]:

![Graphs showing the process for different values of n](image-url)
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.

### 17.5.5 The Multivariate Case

The law of large numbers and central limit theorem work just as nicely in multidimensional settings.

To state the results, let’s recall some elementary facts about random vectors.

A random vector \( \mathbf{X} \) is just a sequence of \( k \) random variables \((X_1, \ldots, X_k)\).

Each realization of \( \mathbf{X} \) is an element of \( \mathbb{R}^k \).

A collection of random vectors \( \mathbf{X}_1, \ldots, \mathbf{X}_n \) is called independent if, given any \( n \) vectors \( \mathbf{x}_1, \ldots, \mathbf{x}_n \) in \( \mathbb{R}^k \), we have

\[
\mathbb{P}\{\mathbf{X}_1 \leq \mathbf{x}_1, \ldots, \mathbf{X}_n \leq \mathbf{x}_n\} = \mathbb{P}\{\mathbf{X}_1 \leq \mathbf{x}_1\} \times \cdots \times \mathbb{P}\{\mathbf{X}_n \leq \mathbf{x}_n\}
\]

(The vector inequality \( \mathbf{X} \leq \mathbf{x} \) means that \( X_j \leq x_j \) for \( j = 1, \ldots, k \))

Let \( \mu_j := \mathbb{E}[X_j] \) for all \( j = 1, \ldots, k \).

The expectation \( \mathbb{E}[\mathbf{X}] \) of \( \mathbf{X} \) is defined to be the vector of expectations:

\[
\mathbb{E}[\mathbf{X}] := \begin{pmatrix} \mathbb{E}[X_1] \\ \mathbb{E}[X_2] \\ \vdots \\ \mathbb{E}[X_k] \end{pmatrix} = \begin{pmatrix} \mu_1 \\ \mu_2 \\ \vdots \\ \mu_k \end{pmatrix} =: \mu
\]

The variance-covariance matrix of random vector \( \mathbf{X} \) is defined as

\[
\text{Var}[\mathbf{X}] := \mathbb{E}[(\mathbf{X} - \mu)(\mathbf{X} - \mu)']
\]

Expanding this out, we get

\[
\text{Var}[\mathbf{X}] = \begin{pmatrix} \mathbb{E}[(X_1 - \mu_1)(X_1 - \mu_1)] & \cdots & \mathbb{E}[(X_1 - \mu_1)(X_k - \mu_k)] \\ \mathbb{E}[(X_2 - \mu_2)(X_1 - \mu_1)] & \cdots & \mathbb{E}[(X_2 - \mu_2)(X_k - \mu_k)] \\ \vdots & \cdots & \vdots \\ \mathbb{E}[(X_k - \mu_k)(X_1 - \mu_1)] & \cdots & \mathbb{E}[(X_k - \mu_k)(X_k - \mu_k)] \end{pmatrix}
\]

The \( j, k \)-th term is the scalar covariance between \( X_j \) and \( X_k \).

With this notation we can proceed to the multivariate LLN and CLT.

Let \( \mathbf{X}_1, \ldots, \mathbf{X}_n \) be a sequence of independent and identically distributed random vectors, each one taking values in \( \mathbb{R}^k \).

Let \( \mu \) be the vector \( \mathbb{E}[\mathbf{X}_1] \), and let \( \Sigma \) be the variance-covariance matrix of \( \mathbf{X}_i \).

Interpreting vector addition and scalar multiplication in the usual way (i.e., pointwise), let
In this setting, the LLN tells us that

\[ P \{ \bar{X}_n \to \mu \text{ as } n \to \infty \} = 1 \]  

(7)

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,

\[ \sqrt{n}(\bar{X}_n - \mu) \xrightarrow{d} N(0, \Sigma) \quad \text{as } n \to \infty \]  

(8)

17.6 Exercises

17.6.1 Exercise 1

One very useful consequence of the central limit theorem is as follows.

Assume the conditions of the CLT as stated above. If \( g : \mathbb{R} \to \mathbb{R} \) is differentiable at \( \mu \) and \( g'(\mu) \neq 0 \), then

\[ \sqrt{n}(g(\bar{X}_n) - g(\mu)) \xrightarrow{d} N(0, g'(\mu)^2 \sigma^2) \quad \text{as } n \to \infty \]  

(9)

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?

17.6.2 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

$$\sqrt{n}(\bar{X}_n - \mu) \xrightarrow{d} N(0, \Sigma) \quad (10)$$

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 $S$, 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

$$n\|Q(\bar{X}_n - \mu)\|^2 \xrightarrow{d} \chi^2(k) \quad (11)$$

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 (11) with a simulation.

In doing so, let
\[ X_i := \left( \begin{array}{c} W_i \\ U_i + W_i \end{array} \right) \]

where

- each \( W_i \) is an IID draw from the uniform distribution on \([-1,1]\)
- each \( U_i \) is an IID draw from the uniform distribution on \([-2,2]\)
- \( U_i \) and \( W_i \) are independent of each other

Hints:

1. `sqrt(A::AbstractMatrix{<:Number})` 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.

17.7 Solutions

17.7.1 Exercise 1

Here is one solution

```julia
function exercise1(distribution = Uniform(0, π/2); n = 250, k = 10_000, g משמעות = sin, g' = cos)
    μ, σ = mean(distribution), std(distribution)
    y = rand(distribution, n, k)
    y = mean(y, dims = 1)
    y = vec(y)
    error_obs = sqrt(n) .* (g.(y) .- g.(μ))
    density(error_obs, label = "Empirical Density")
    return plot!(Normal(0, g'(μ) .* σ), linestyle = :dash, label = "Asymptotic", color = :black)
end
exercise1()
```

Out[18]:
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.

### 17.7.2 Exercise 2

First we want to verify the claim that

\[
\sqrt{n}Q(\bar{X}_n - \mu) \xrightarrow{d} N(0, I)
\]

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, \(QY_n \xrightarrow{d} QY \sim N(0, I)\), which is what we aimed to show.

Now we turn to the simulation exercise.

Our solution is as follows
In [19]: function exercise2(;n = 250, k = 50_000, dw = Uniform(-1, 1), du = Uniform(-2, 2))
    vw = var(dw)
    vu = var(du)
    Σ = [vw vw
         vw vu + vu]
    Q = inv(sqrt(Σ))
    function generate_data(dw, du, n)
        dw = rand(dw, n)
        X = [dw dw + rand(du, n)]
        return sqrt(n) * mean(X, dims = 1)
    end
    X = mapreduce(x -> generate_data(dw, du, n), vcat, 1:k)
    X = Q * X'
    X = sum(abs2, X, dims = 1)
    X = vec(X)
    density(X, label = "", xlim = (0, 10))
    return plot!(Chisq(2), color = :black, linestyle = :dash,
                label = "Chi-squared with 2 degrees of freedom", grid = false)
end
exercise2()

Out[19]:
Chapter 18

Linear State Space Models

18.1 Contents

- Overview 18.2
- The Linear State Space Model 18.3
- Distributions and Moments 18.4
- Stationarity and Ergodicity 18.5
- Noisy Observations 18.6
- Prediction 18.7
- Code 18.8
- Exercises 18.9
- Solutions 18.10

“We may regard the present state of the universe as the effect of its past and the cause of its future” – Marquis de Laplace

18.2 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
  - Friedman’s permanent income model of consumption smoothing
  - Barro’s model of smoothing total tax collections
  - Rational expectations version of Cagan’s model of hyperinflation
  - Sargent and Wallace’s “unpleasant monetarist arithmetic,” etc.
18.2.1 Setup

In [1]: using InstantiateFromURL
   # optionally add arguments to force installation: instantiate = true, 
   precompile = true
   github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

In [2]: using LinearAlgebra, Statistics

18.3 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

$$
\begin{align*}
x_{t+1} &= Ax_t + Cw_{t+1} \\
y_t &= Gx_t \\
x_0 &\sim N(\mu_0, \Sigma_0)
\end{align*}
$$

18.3.1 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 (1) pins down the values of the sequences $\{x_t\}$ and $\{y_t\}$.

Even without these draws, the primitives 1–3 pin down the probability distributions of $\{x_t\}$ and $\{y_t\}$.

Later we’ll see how to compute these distributions and their moments.

Martingale difference shocks

We’ve made the common assumption that the shocks are independent standardized normal vectors.

But some of what we say will be valid under the assumption that $\{w_{t+1}\}$ is a martingale difference sequence.
A martingale difference sequence is a sequence that is zero mean when conditioned on past information.

In the present case, since \( \{x_t\} \) is our state sequence, this means that it satisfies

\[
\mathbb{E}[w_{t+1}|x_t, x_{t-1}, \ldots] = 0
\]

This is a weaker condition than that \( \{w_t\} \) is iid with \( w_{t+1} \sim N(0, I) \).

### 18.3.2 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

\[
y_{t+1} = \phi_0 + \phi_1 y_t + \phi_2 y_{t-1} \quad \text{s.t.} \quad y_0, y_{-1} \text{ given}
\]

(2)

To map (2) into our state space system (1), we set

\[
x_t = \begin{bmatrix} 1 \\ y_t \\ y_{t-1} \end{bmatrix}, \quad A = \begin{bmatrix} 1 & 0 & 0 \\ \phi_0 & \phi_1 & \phi_2 \\ 0 & 1 & 0 \end{bmatrix}, \quad C = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \quad G = [0 \ 1 \ 0]
\]

You can confirm that under these definitions, (1) and (2) 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 (1) to represent the model

\[ y_{t+1} = \phi_1 y_t + \phi_2 y_{t-1} + \phi_3 y_{t-2} + \phi_4 y_{t-3} + \sigma w_{t+1} \]  

(3)

where \( \{w_t\} \) is iid and standard normal.

To put this in the linear state space format we take

\[ x_t = \begin{bmatrix} y_t \\ y_{t-1} \\ y_{t-2} \\ y_{t-3} \end{bmatrix} \]

and

\[ A = \begin{bmatrix} \phi_1 & \phi_2 & \phi_3 & \phi_4 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \]

\[ C = \begin{bmatrix} \sigma \\ 0 \\ 0 \\ 0 \end{bmatrix} \]

\[ G = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix} \]

The matrix \( A \) has the form of the *companion matrix* to the vector \( [\phi_1 \ \phi_2 \ \phi_3 \ \phi_4] \).

The next figure shows dynamics of this process when

\[ \phi_1 = 0.5, \ \phi_2 = -0.2, \ \phi_3 = 0, \ \phi_4 = 0.5, \ \sigma = 0.2, \ y_0 = y_{-1} = y_{-2} = y_{-3} = 1 \]

---

**Vector Autoregressions**

Now suppose that

- \( y_t \) is a \( k \times 1 \) vector
- \( \phi_j \) is a \( k \times k \) matrix and
- \( w_t \) is \( k \times 1 \)

Then (3) is termed a *vector autoregression*. 
To map this into (1), 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 (1) 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 (3).

With the deterministic seasonal, the transition matrix becomes

\[ A = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \]

It is easy to check that \( A^4 = I \), which implies that \( x_t \) is strictly periodic with period 4:

\[ x_{t+4} = x_t \]

Such an \( x_t \) process can be used to model deterministic seasonals in quarterly time series.

The *indeterministic seasonal* produces recurrent, but aperiodic, seasonal fluctuations.

**Time Trends**

The model \( y_t = at + b \) is known as a *linear time trend*.

We can represent this model in the linear state space form by taking

\[ A = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \quad C = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad G = \begin{bmatrix} a & b \end{bmatrix} \]

(4)

and starting at initial condition \( x_0 = [0 \ 1]' \).

In fact it’s possible to use the state-space system to represent polynomial trends of any order. For instance, let
\[
x_0 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \quad A = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \quad C = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}
\]

It follows that
\[
A^t = \begin{bmatrix} 1 & t & t(t-1)/2 \\ 0 & 1 & t \\ 0 & 0 & 1 \end{bmatrix}
\]

Then \(x'_t = [t(t-1)/2 \ t \ 1]\), so that \(x_t\) contains linear and quadratic time trends.

### 18.3.3 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 (1) repeatedly to obtain

\[
x_t = Ax_{t-1} + Cw_t
= A^2x_{t-2} + ACw_{t-1} + Cw_t
\]

\[
\vdots
= \sum_{j=0}^{t-1} A^jCw_{t-j} + A^t x_0
\]

Representation (5) is a moving average representation. It expresses \(\{x_t\}\) as a linear function of

1. current and past values of the process \(\{w_t\}\) and
2. the initial condition \(x_0\)

As an example of a moving average representation, let the model be

\[
A = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \quad C = \begin{bmatrix} 1 \\ 0 \end{bmatrix}
\]

You will be able to show that \(A^t = \begin{bmatrix} 1 & t \\ 0 & 1 \end{bmatrix}\) and \(A^iC = [1 \ 0]'.\)

Substituting into the moving average representation (5), we obtain

\[
x_{1t} = \sum_{j=0}^{t-1} w_{t-j} + [1 \ t] x_0
\]

where \(x_{1t}\) is the first entry of \(x_t\).

The first term on the right is a cumulated sum of martingale differences, and is therefore a martingale.

The second term is a translated linear function of time.

For this reason, \(x_{1t}\) is called a martingale with drift.
18.4 Distributions and Moments

18.4.1 Unconditional Moments

Using (1), it’s easy to obtain expressions for the (unconditional) means of \( x_t \) and \( y_t \).

We’ll explain what unconditional and conditional mean soon.

Letting \( \mu_t := \mathbb{E}[x_t] \) and using linearity of expectations, we find that

\[
\mu_{t+1} = A \mu_t \quad \text{with} \quad \mu_0 \text{ given}
\]  

(6)

Here \( \mu_0 \) is a primitive given in (1).

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

\[
\Sigma_{t+1} = A \Sigma_t A' + CC' \quad \text{with} \quad \Sigma_0 \text{ given}
\]  

(7)

As with \( \mu_0 \), the matrix \( \Sigma_0 \) is a primitive given in (1).

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

\[
\mathbb{E}[y_t] = \mathbb{E}[G x_t] = G \mu_t
\]  

(8)

The variance-covariance matrix of \( y_t \) is easily shown to be

\[
\text{Var}[y_t] = \text{Var}[G x_t] = G \Sigma_t G'
\]  

(9)

18.4.2 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

\[ u \sim N(\bar{u}, S) \quad \text{and} \quad v = a + Bu \implies v \sim N(a + B\bar{u}, BSB') \]  
(10)

In particular, given our Gaussian assumptions on the primitives and the linearity of (1) we can see immediately that both \( x_t \) and \( y_t \) are Gaussian for all \( t \geq 0 \). Section ??.

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 (6) and (7).

Letting \( \mu_t \) and \( \Sigma_t \) be as defined by these equations, we have

\[ x_t \sim N(\mu_t, \Sigma_t) \]  
(11)

By similar reasoning combined with (8) and (9),

\[ y_t \sim N(G\mu_t, G\Sigma_t G') \]  
(12)

18.4.3 Ensemble Interpretations

How should we interpret the distributions defined by (11)–(12)?

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 (3).

The values of \( y_T \) are represented by black dots in the left-hand figure.
In the right-hand figure, these values are converted into a rotated histogram that shows relative frequencies from our sample of 20 $y_T$’s.

(The parameters and source code for the figures can be found in file `linear_models/paths_and_hist.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 (12).
CHAPTER 18. LINEAR STATE SPACE MODELS

The histogram and population distribution are close, as expected.

By looking at the figures and experimenting with parameters, you will gain a feel for how the population distribution depends on the model primitives listed above, as intermediated by the distribution’s sufficient statistics.

**Ensemble means**

In the preceding figure we approximated the population distribution of \( y_T \) by

1. generating \( I \) sample paths (i.e., time series) where \( I \) is a large number
2. recording each observation \( y^i_T \)
3. histogramming this sample

Just as the histogram approximates the population distribution, the *ensemble* or *cross-sectional average*

\[
\bar{y}_T := \frac{1}{I} \sum_{i=1}^{I} y^i_T
\]

approximates the expectation \( \mathbb{E}[y_T] = G\mu_T \) (as implied by the law of large numbers).

Here’s a simulation comparing the ensemble averages and population means at time points \( t = 0, \ldots, 50 \).

The parameters are the same as for the preceding figures, and the sample size is relatively small (\( I = 20 \)).

The ensemble mean for \( x_t \) is

\[
\bar{x}_T := \frac{1}{I} \sum_{i=1}^{I} x^i_T \rightarrow \mu_T \quad (I \rightarrow \infty)
\]
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

$$
\frac{1}{I} \sum_{i=1}^{I} (x_T^i - \bar{x}_T)(x_T^i - \bar{x}_T)' \rightarrow \Sigma_T \quad (I \rightarrow \infty)
$$

18.4.4 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 (1), 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

$$\Sigma_{t+j, t} := \mathbb{E}[(x_{t+j} - \mu_{t+j})(x_t - \mu_t)']$$

Elementary calculations show that

$$\Sigma_{t+j, t} = A^j \Sigma_t$$
Notice that $\Sigma_{t+j,t}$ in general depends on both $j$, the gap between the two dates, and $t$, the earlier date.

### 18.5 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.

#### 18.5.1 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*.

#### 18.5.2 Stationary Distributions

In our setting, a distribution $\psi_\infty$ is said to be *stationary* for $x_t$ if

$$x_t \sim \psi_\infty \quad \text{and} \quad x_{t+1} = Ax_t + C w_{t+1} \implies x_{t+1} \sim \psi_\infty$$

Since

1. in the present case all distributions are Gaussian
2. A Gaussian distribution is pinned down by its mean and variance-covariance matrix we can restate the definition as follows: \( \psi_\infty \) is stationary for \( x_t \) if

\[
\psi_\infty = N(\mu_\infty, \Sigma_\infty)
\]

where \( \mu_\infty \) and \( \Sigma_\infty \) are fixed points of (6) and (7) respectively.

### 18.5.3 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 (6) and (7) respectively

we’ve ensured that

\[
\mu_t = \mu_\infty \quad \text{and} \quad \Sigma_t = \Sigma_\infty \quad \text{for all} \ t
\]

Moreover, in view of (14), 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 \( t \).
18.5.4 Conditions for Stationarity

The globally stable case

The difference equation $\mu_{t+1} = A\mu_t$ is known to have unique fixed point $\mu_\infty = 0$ if all eigenvalues of $A$ have moduli strictly less than unity. That is, if $\text{all(abs(eigvals(A))} < 1) == \text{true}$.

The difference equation (7) also has a unique fixed point in this case, and, moreover

$$\mu_t \to \mu_\infty = 0 \quad \text{and} \quad \Sigma_t \to \Sigma_\infty \quad \text{as} \quad t \to \infty$$

regardless of the initial conditions $\mu_0$ and $\Sigma_0$.

This is the globally stable case — see these notes for more a theoretical treatment.

However, global stability is more than we need for stationary solutions, and often more than we want.

To illustrate, consider our second order difference equation example.

Here the state is $x_t = \begin{bmatrix} 1 & y_t & y_{t-1} \end{bmatrix}'$.

Because of the constant first component in the state vector, we will never have $\mu_t \to 0$.

How can we find stationary solutions that respect a constant state component?

Processes with a constant state component

To investigate such a process, suppose that $A$ and $C$ take the form

$$A = \begin{bmatrix} A_1 & a \\ 0 & 1 \end{bmatrix} \quad C = \begin{bmatrix} C_1 \\ 0 \end{bmatrix}$$

where

- $A_1$ is an $(n - 1) \times (n - 1)$ matrix
- $a$ is an $(n - 1) \times 1$ column vector

Let $x_t = \begin{bmatrix} x_{1t}' & 1 \end{bmatrix}'$ where $x_{1t}$ is $(n - 1) \times 1$.

It follows that

$$x_{1,t+1} = A_1x_{1t} + a + C_1w_{t+1}$$

Let $\mu_{1t} = \mathbb{E}[x_{1t}]$ and take expectations on both sides of this expression to get

$$\mu_{1,t+1} = A_1\mu_{1t} + a \quad \quad (15)$$

Assume now that the moduli of the eigenvalues of $A_1$ are all strictly less than one.

Then (15) has a unique stationary solution, namely,

$$\mu_{1\infty} = (I - A_1)^{-1}a$$
The stationary value of $\mu_t$ itself is then $\mu_\infty := [\mu'_\infty \ 1]'$.

The stationary values of $\Sigma_t$ and $\Sigma_{t+j,t}$ satisfy

$$\Sigma_\infty = A \Sigma_\infty A' + CC'$$
$$\Sigma_{t+j,t} = A^j \Sigma_\infty$$

(16)

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 (7) converge to the fixed point of the *discrete Lyapunov equation* in the first line of (16).

### 18.5.5 Ergodicity

Let’s suppose that we’re working with a covariance stationary process.

In this case we know that the ensemble mean will converge to $\mu_\infty$ as the sample size $I$ approaches infinity.

**Averages over time**

Ensemble averages across simulations are interesting theoretically, but in real life we usually observe only a single realization $\{x_t, y_t\}_{t=0}^T$.

So now let’s take a single realization and form the time series averages

$$\bar{x} := \frac{1}{T} \sum_{t=1}^T x_t \quad \text{and} \quad \bar{y} := \frac{1}{T} \sum_{t=1}^T y_t$$

Do these time series averages converge to something interpretable in terms of our basic state-space representation?

The answer depends on something called *ergodicity*.

Ergodicity is the property that time series and ensemble averages coincide.

More formally, ergodicity implies that time series sample averages converge to their expectation under the stationary distribution.

In particular,

- $\frac{1}{T} \sum_{t=1}^T x_t \rightarrow \mu_\infty$
- $\frac{1}{T} \sum_{t=1}^T (x_t - \bar{x}_T)(x_t - \bar{x}_T)' \rightarrow \Sigma_\infty$
- $\frac{1}{T} \sum_{t=1}^T (x_{t+j} - \bar{x}_T)(x_t - \bar{x}_T)' \rightarrow A^j \Sigma_\infty$
In our linear Gaussian setting, any covariance stationary process is also ergodic.

18.6 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

\[
\begin{align*}
x_{t+1} &= Ax_t + Cw_{t+1} \\
y_t &= Gx_t + Hv_t \\
x_0 &\sim N(\mu_0, \Sigma_0)
\end{align*}
\] (17)

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 (8) and (9) now become

\[
\mathbb{E}[y_t] = \mathbb{E}[Gx_t + Hv_t] = G\mu_t
\] (18)

The variance-covariance matrix of \( y_t \) is easily shown to be

\[
\text{Var}[y_t] = \text{Var}[Gx_t + Hv_t] = G\Sigma_t G' + HH'
\] (19)

The distribution of \( y_t \) is therefore

\[
y_t \sim N(G\mu_t, G\Sigma_t G' + HH')
\]

18.7 Prediction

The theory of prediction for linear state space systems is elegant and simple.

18.7.1 Forecasting Formulas – Conditional Means

The natural way to predict variables is to use conditional distributions.

For example, the optimal forecast of \( x_{t+1} \) given information known at time \( t \) is

\[
\mathbb{E}_t[x_{t+1}] := \mathbb{E}[x_{t+1} | x_t, x_{t-1}, \ldots, x_0] = Ax_t
\]

The right-hand side follows from \( x_{t+1} = Ax_t + Cw_{t+1} \) and the fact that \( w_{t+1} \) is zero mean and independent of \( x_t, x_{t-1}, \ldots, x_0 \).
That $\mathbb{E}_t[x_{t+1} | x_t]$ is an implication of $\{x_t\}$ having the Markov property.

The one-step-ahead forecast error is

$$x_{t+1} - \mathbb{E}_t[x_{t+1}] = Cw_{t+1}$$

The covariance matrix of the forecast error is

$$\mathbb{E}[(x_{t+1} - \mathbb{E}_t[x_{t+1}])(x_{t+1} - \mathbb{E}_t[x_{t+1}])'] = CC'$$

More generally, we’d like to compute the $j$-step ahead forecasts $\mathbb{E}_t[x_{t+j}]$ and $\mathbb{E}_t[y_{t+j}]$.

With a bit of algebra we obtain

$$x_{t+j} = A^j x_t + A^{j-1} C w_{t+1} + A^{j-2} C w_{t+2} + \cdots + A^0 C w_{t+j}$$

In view of the iid property, current and past state values provide no information about future values of the shock.

Hence $\mathbb{E}_t[w_{t+k}] = \mathbb{E}[w_{t+k}] = 0$.

It now follows from linearity of expectations that the $j$-step ahead forecast of $x$ is

$$\mathbb{E}_t[x_{t+j}] = A^j x_t$$

The $j$-step ahead forecast of $y$ is therefore

$$\mathbb{E}_t[y_{t+j}] = \mathbb{E}_t[Gx_{t+j} + Hv_{t+j}] = GA^j x_t$$

### 18.7.2 Covariance of Prediction Errors

It is useful to obtain the covariance matrix of the vector of $j$-step-ahead prediction errors

$$x_{t+j} - \mathbb{E}_t[x_{t+j}] = \sum_{s=0}^{j-1} A^s C w_{t-s+j}$$

Evidently,

$$V_j := \mathbb{E}_t[(x_{t+j} - \mathbb{E}_t[x_{t+j}])(x_{t+j} - \mathbb{E}_t[x_{t+j}])'] = \sum_{k=0}^{j-1} A^k CC' A^k'$$

$V_j$ defined in (21) can be calculated recursively via $V_1 = CC'$ and

$$V_j = CC' + AV_{j-1}A', \quad j \geq 2$$

$V_j$ is the conditional covariance matrix of the errors in forecasting $x_{t+j}$, conditioned on time $t$ information $x_t$.

Under particular conditions, $V_j$ converges to
\[ V_\infty = CC' + AV_\infty A' \quad (23) \]

Equation (23) 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.

### 18.7.3 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 (1).

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
  \[
  \mathbb{E}_t \left[ \sum_{j=0}^{\infty} \beta^j y_{t+j} \right] = G[I + \beta A + \beta^2 A^2 + \cdots] x_t = G[I - \beta A]^{-1} x_t
  \]

### 18.8 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.

18.9 Exercises

18.9.1 Exercise 1

Replicate this figure using the LSS type from lss.jl.

18.9.2 Exercise 2

Replicate this figure modulo randomness using the same type.

18.9.3 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.

18.9.4 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.

18.10 Solutions

In [3]: using QuantEcon, Plots
   gr(fmt=:png);

18.10.1 Exercise 1

In [4]: ϕ₀, ϕ₁, ϕ₂ = 1.1, 0.8, -0.8
   A = [1.0 0.0 0
        ϕ₀ ϕ₁ ϕ₂
        0.0 1.0 0.0]
   C = zeros(3, 1)
   G = [0.0 1.0 0.0]
   μ₀ = ones(3)
\texttt{18.10.2 Exercise 2}

\textbf{In [5]:} \texttt{using Random}  
\texttt{Random.seed!}(42) \texttt{# For deterministic results.}

\begin{align*}
\phi_1, \phi_2, \phi_3, \phi_4 & = 0.5, -0.2, 0, 0.5 \\
\sigma & = 0.2 \\
A & = \begin{bmatrix}
\phi_1 & \phi_2 & \phi_3 & \phi_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 & = \text{LSS}(A, C, G; \mu_0 = \text{ones}(4)) \\
x, y & = \text{simulate}(ar, 200) \\
\text{plot}(\text{dropdims}(y, \text{dims} = 1), \text{color} = \text{:blue}, \text{linewidth} = 2, \text{alpha} = 0.7) \\
\text{plot!}(!\text{xlabel}="time", !\text{ylabel} = "y_t", !\text{legend} = \text{:none})
\end{align*}

\textbf{Out[5]:}
18.10.3 Exercise 3

In [6]: $\phi_1, \phi_2, \phi_3, \phi_4 = 0.5, -0.2, 0, 0.5$
$\sigma = 0.1$

$A = \begin{bmatrix} \phi_1 & \phi_2 & \phi_3 & \phi_4 \\ 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \end{bmatrix}$

$C = \begin{bmatrix} \sigma \\ 0.0 \\ 0.0 \end{bmatrix}$

$G = [1.0, 0.0, 0.0, 0.0]$
$I = 20$
$T = 50$
$ar = \text{LSS}(A, C, G; \text{mu}_0 = \text{ones}(4))$
$y_{\text{min}}, y_{\text{max}} = -0.5, 1.15$

ensemble_mean = zeros(T)
y_s = []
for i in 1:I
    x, y = simulate(ar, T)
y = dropdims(y, dims = 1)
push!(y_s, y)
    ensemble_mean .+= y
end

ensemble_mean = ensemble_mean ./ I
plot(y_s, color = :blue, alpha = 0.2, linewidth = 0.8, label = "")
plot!(ensemble_mean, color = :blue, linewidth = 2, label = "y_{t\_bar}"

m = moment_sequence(ar)
pop_means = zeros(0)
for (i, t) in enumerate(m)
    \((\mu_x, \mu_y, \Sigma_x, \Sigma_y) = t\)
    push!(pop_means, \(\mu_y[1]\))
    i == 50 && break
end
plot!(pop_means, color = :green, linewidth = 2, label = "G mu_t")
plot!(ylims=(ymin, ymax), xlabel = "time", ylabel = "y_t", legendfont = [:font(12)])

Out[6]:

18.10.4 Exercise 4

In [7]: \(\phi_1, \phi_2, \phi_3, \phi_4 = 0.5, -0.2, 0, 0.5\)
\(\sigma = 0.1\)
\[A = [\phi_1 \phi_2 \phi_3 \phi_4]
   \begin{bmatrix}
   1.0 & 0.0 & 0.0 & 0.0 \\
   0.0 & 1.0 & 0.0 & 0.0 \\
   0.0 & 0.0 & 1.0 & 0.0 \\
   \end{bmatrix}\]
\[C = [\sigma \sigma \sigma \sigma]^T\]
\[G = [1.0 0.0 0.0 0.0]^T\]
\[T_0 = 10\]
\[T_1 = 50\]
\[T_2 = 75\]
\[T_4 = 100\]
\[ar = \text{LSS}(A, C, G; mu_0 = \text{ones}(4))\]
$\text{ys} = []$
$x\_\text{scatter} = []$
$y\_\text{scatter} = []$

for $i \equiv 1:80$
    $r\text{color} = \text{colors}[\text{rand}(1:3)]$
    $x, y = \text{simulate}(ar, T4)$
    $y = \text{dropdims}(y, \text{dims} = 1)$
    $\text{push!}(\text{ys}, y)$
    $x\_\text{scatter} = [x\_\text{scatter}; T0; T1; T2]$
    $y\_\text{scatter} = [y\_\text{scatter}; y[T0]; y[T1]; y[T2]]$
end

plot(ys, linewidth = 0.8, alpha = 0.5)
plot!([T0 T1 T2; T0 T1 T2], [-1 -1 -1; 1 1 1], color = :black, legend = :none)
scatter!(x\_\text{scatter}, y\_\text{scatter}, color = :black, alpha = 0.5)
plot!(ylims = (ymin, ymax), ylabel = "y$_t$", xticks = [], yticks = ymin:0.2:ymax)
plot!(annotations = [(T0+1, -0.55, "T");(T1+1, -0.55, "T'");(T2+1, -0.55, "T''")])

Footnotes

[1] The eigenvalues of $A$ are $(1, -1, i, -i)$.
The correct way to argue this is by induction. Suppose that $x_t$ is Gaussian. Then (1) and (10) 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.
Chapter 19

Finite Markov Chains

19.1 Contents

- Overview 19.2
- Definitions 19.3
- Simulation 19.4
- Marginal Distributions 19.5
- Irreducibility and Aperiodicity 19.6
- Stationary Distributions 19.7
- Ergodicity 19.8
- Computing Expectations 19.9
- Exercises 19.10
- Solutions 19.11

19.2 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.

19.2.1 Setup

In [1]: using InstantiateFromURL
   # optionally add arguments to force installation: instantiate = true, 
   # precompile = true
   github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")
In [2]: using LinearAlgebra, Statistics
       using Distributions, Plots, Printf, QuantEcon, Random
       gr(fmt = :png);

19.3 Definitions

The following concepts are fundamental.

19.3.1 Stochastic Matrices

A stochastic matrix (or Markov matrix) is an \( n \times n \) square matrix \( P \) such that

1. each element of \( P \) is nonnegative, and
2. each row of \( P \) sums to one

Each row of \( P \) can be regarded as a probability mass function over \( n \) possible outcomes. It is too not difficult to check Section ?? that if \( P \) is a stochastic matrix, then so is the \( k \)-th power \( P^k \) for all \( k \in \mathbb{N} \).

19.3.2 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 \( y \in S \),

\[
\mathbb{P}\{X_{t+1} = y \mid X_t\} = \mathbb{P}\{X_{t+1} = y \mid X_t, X_{t-1}, \ldots\} \tag{1}
\]

In other words, knowing the current state is enough to know probabilities for future states. In particular, the dynamics of a Markov chain are fully determined by the set of values

\[
P(x, y) := \mathbb{P}\{X_{t+1} = y \mid X_t = x\} \quad (x, y \in S) \tag{2}
\]

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).

19.3.3 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?

We’ll cover such applications below.

19.3.4 Example 2

Using US unemployment data, Hamilton [38] estimated the stochastic matrix

$$P = \begin{pmatrix} 0.971 & 0.029 & 0 \\ 0.145 & 0.778 & 0.077 \\ 0 & 0.508 & 0.492 \end{pmatrix}$$

where

- the frequency is monthly
- the first state represents “normal growth”
- the second state represents “mild recession”
- the third state represents “severe recession”

For example, the matrix tells us that when the state is normal growth, the state will again be normal growth next month with probability 0.97.

In general, large values on the main diagonal indicate persistence in the process $\{X_t\}$. 
This Markov process can also be represented as a directed graph, with edges labeled by transition probabilities

Here “ng” is normal growth, “mr” is mild recession, etc.

19.4 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, it’s also a good exercise to roll our own routines — let’s do that first and then come back to the methods in QuantEcon.jl.

In these exercises we’ll take the state space to be \( S = 1, \ldots, n \).

19.4.1 Rolling our own

To simulate a Markov chain, we need its stochastic matrix \( P \) and either an initial state or a probability distribution \( \psi \) for initial state to be drawn from.

The Markov chain is then constructed as discussed above. To repeat:

1. At time \( t = 0 \), the \( X_0 \) is set to some fixed state or chosen from \( \psi \).
2. At each subsequent time \( t \), the new state \( X_{t+1} \) is drawn from \( P(X_t, \cdot) \).

In order to implement this simulation procedure, we need a method for generating draws from a discrete distribution.

For this task we’ll use a Categorical random variable (i.e. a discrete random variable with assigned probabilities)

In [3]: d = Categorical([0.5, 0.3, 0.2])  # 3 discrete states
   @show rand(d, 5)
   @show supertype(typeof(d))
   @show pdf(d, 1)  # the probability to be in state 1
   @show support(d)
   @show pdf.(d, support(d));  # broadcast the pdf over the whole support
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 $\text{sample\_size}$ representing the length of the time series the function should return

```python
In [4]: function mc_sample_path(P; init = 1, sample_size = 1000)
   @assert size(P)[1] == size(P)[2] # square required
   N = size(P)[1] # should be square

   # create vector of discrete RVs for each row
dists = [Categorical(P[i, :]) for i in 1:N]

   # setup the simulation
   X = fill(0, sample_size) # allocate memory, or zeros(Int64, sample_size)
   X[1] = init # set the initial state

   for t in 2:sample_size
       dist = dists[X[t-1]] # get discrete RV from last state's transition
       X[t] = rand(dist) # draw new value
   end
   return X
end
```

```
Out[4]: mc_sample_path (generic function with 1 method)
```

Let’s see how it works using the small matrix

$$P := \begin{pmatrix} 0.4 & 0.6 \\ 0.2 & 0.8 \end{pmatrix}$$

(3)

As we’ll see later, for a long series drawn from $P$, the fraction of the sample that takes value 1 will be about 0.25.

If you run the following code you should get roughly that answer

```python
In [5]: P = [0.4, 0.6; 0.2, 0.8]
   X = mc_sample_path(P, sample_size = 100_000); # note 100_000 = 100000
   μ₁ = count(X .== 1)/length(X) # .== broadcasts test for equality. Could
   use mean(X .== 1)

Out[5]: 0.25014
```
19.4.2 Using QuantEcon’s Routines

As discussed above, QuantEcon.jl has routines for handling Markov chains, including simulation.

Here’s an illustration using the same P as the preceding example

In [6]:
`P = [0.4 0.6; 0.2 0.8];`  
`mc = MarkovChain(P)`  
`X = simulate(mc, 100_000);`  
`μ_2 = count(X .== 1)/length(X) # or mean(x -> x == 1, X)`

Out[6]: 0.25053

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

In [7]:
`mc = MarkovChain(P, ["unemployed", "employed"])`  
`simulate(mc, 4, init = 1) # start at state 1`

Out[7]: 4-element Array{String,1}:
"unemployed"
"unemployed"
"employed"
"employed"

In [8]: `simulate(mc, 4, init = 2) # start at state 2`

Out[8]: 4-element Array{String,1}:
"employed"
"employed"
"employed"
"employed"

In [9]: `simulate(mc, 4) # start with randomly chosen initial condition`

Out[9]: 4-element Array{String,1}:
"employed"
"unemployed"
"unemployed"
"unemployed"

In [10]: `simulate_indices(mc, 4)`

Out[10]: 4-element Array{Int64,1}:
2
2
2
2
19.5 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 \( X_{t+m} \)?

19.5.1 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.

\[
\psi_{t+1} = \psi_t P
\]

In other words, to move the distribution forward one unit of time, we postmultiply by \( P \).

By repeating this \( m \) times we move forward \( m \) steps into the future.

Hence, iterating on (4), the expression \( \psi_{t+m} = \psi_t P^m \) is also valid — here \( P^m \) is the \( m \)-th power of \( P \).

As a special case, we see that if \( \psi_0 \) is the initial distribution from which \( X_0 \) is drawn, then \( \psi_0 P^m \) is the distribution of \( X_m \).

This is very important, so let’s repeat it

\[
X_0 \sim \psi_0 \implies X_m \sim \psi_0 P^m \quad (5)
\]

and, more generally,

\[
X_t \sim \psi_t \implies X_{t+m} \sim \psi_t P^m \quad (6)
\]
19.5.2 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 (6), but now with \( \psi_t \) putting all probability on state \( x \).

- 1 in the \( x \)-th position and zero elsewhere.

Inserting this into (6), we see that, conditional on \( X_t = x \), the distribution of \( X_{t+m} \) is the \( x \)-th row of \( P^m \).

In particular

\[
\mathbb{P}\{X_{t+m} = y\} = P^m(x, y) = (x, y)\text{-th element of } P^m
\]

19.5.3 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

\[
\psi P^6 \cdot \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}
\]

19.5.4 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 worker’s career spent being employed and unemployed, respectively.
19.6 Irreducibility and Aperiodicity

Irreducibility and aperiodicity are central concepts of modern Markov chain theory. Let’s see what they’re about.

19.6.1 Irreducibility

Let $P$ be a fixed stochastic matrix. Two states $x$ and $y$ are said to communicate with each other if there exist positive integers $j$ and $k$ such that

$$P^j(x, y) > 0 \quad \text{and} \quad P^k(y, x) > 0$$

In view of our discussion above, this means precisely that

- state $x$ can be reached eventually from state $y$, and
- state $y$ can be reached eventually from state $x$

The stochastic matrix $P$ is called irreducible if all states communicate; that is, if $x$ and $y$ communicate for all $(x, y)$ in $S \times S$.

For example, consider the following transition probabilities for wealth of a fictitious set of households

We can translate this into a stochastic matrix, putting zeros where there’s no edge between nodes

$$P := \begin{pmatrix} 0.9 & 0.1 & 0 \\ 0.4 & 0.4 & 0.2 \\ 0.1 & 0.1 & 0.8 \end{pmatrix}$$

It’s clear from the graph that this stochastic matrix is irreducible: we can reach any state from any other state eventually.

We can also test this using QuantEcon.jl’s MarkovChain class

```
In [11]: 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)

Out[11]: true
```
Here’s a more pessimistic scenario, where the poor are poor forever

This stochastic matrix is not irreducible, since, for example, rich is not accessible from poor.

Let’s confirm this

```
In [12]: 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)
```

```
Out[12]: false
```

We can also determine the “communication classes,” or the sets of communicating states (where communication refers to a nonzero probability of moving in each direction).

```
In [13]: communication_classes(mc)
```

```
Out[13]: 2-element Array{Array{Int64,1},1}:
  [1]
  [2, 3]
```

It might be clear to you already that irreducibility is going to be important in terms of long run outcomes.

For example, poverty is a life sentence in the second graph but not the first.

We’ll come back to this a bit later.

19.6.2 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:

\[
\begin{pmatrix}
0 & 1 & 0 \\
0 & 0 & 1 \\
1 & 0 & 0
\end{pmatrix}
\]

\[
mc = \text{MarkovChain}(P);
\]

\[
\text{period}(mc)
\]

Out[14]: 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

\[
\begin{pmatrix}
1.0 \\
0.5 \\
0.5 \\
0.5
\end{pmatrix}
\]

We can confirm that the stochastic matrix is periodic as follows

\[
\begin{pmatrix}
0 & 1 & 0 \\
0 & 0 & 1 \\
1 & 0 & 0
\end{pmatrix}
\]

\[
mc = \text{MarkovChain}(P);
\]

\[
\text{period}(mc)
\]

Out[15]: 2

\[
\text{is_aperiodic}(mc)
\]

Out[16]: false
19.7 Stationary Distributions

As seen in (4), we can shift probabilities forward one unit of time via postmultiplication by \( P \).

Some distributions are invariant under this updating process — for example,

\[
\begin{bmatrix}
0.4 & 0.6 \\
0.2 & 0.8
\end{bmatrix};
\]

\[
\begin{bmatrix}
0.25 \\
0.75
\end{bmatrix};
\]

\[
\begin{bmatrix}
0.75 \\
0.25
\end{bmatrix}
\]

In [17]: \( P = [.4 \; .6; \; .2 \; .8] \);
\[
\psi = [0.25, \; 0.75];
\]

\[
\psi^* \times P
\]

Out[17]: 1x2 Adjoint{Float64,Array{Float64,1}}:
\[
\begin{bmatrix}
0.25 \\
0.75
\end{bmatrix}
\]

Such distributions are called stationary, or invariant.

Formally, a distribution \( \psi^* \) on \( S \) is called stationary for \( P \) if \( \psi^* = \psi^* P \).

From this equality we immediately get \( \psi^* = \psi^* P^t \) for all \( t \).

This tells us an important fact: If the distribution of \( X_0 \) is a stationary distribution, then \( X_t \) will have this same distribution for all \( t \).

Hence stationary distributions have a natural interpretation as stochastic steady states — we’ll discuss this more in just a moment.

Mathematically, a stationary distribution is a fixed point of \( P \) when \( P \) is thought of as the map \( \psi \mapsto \psi P \) from (row) vectors to (row) vectors.

**Theorem.** Every stochastic matrix \( P \) has at least one stationary distribution.

(We are assuming here that the state space \( S \) is finite; if not more assumptions are required)

For a proof of this result you can apply Brouwer’s fixed point theorem, or see EDTC, theorem 4.3.5.

There may in fact be many stationary distributions corresponding to a given stochastic matrix \( P \).

- For example, if \( P \) is the identity matrix, then all distributions are stationary.

Since stationary distributions are long run equilibria, to get uniqueness we require that initial conditions are not infinitely persistent.

Infinite persistence of initial conditions occurs if certain regions of the state space cannot be accessed from other regions, which is the opposite of irreducibility.

This gives some intuition for the following fundamental theorem.

**Theorem.** If \( P \) is both aperiodic and irreducible, then

1. \( P \) has exactly one stationary distribution \( \psi^* \).

2. For any initial distribution \( \psi_0 \), we have \( \|\psi_0 P^t - \psi^*\| \to 0 \) as \( t \to \infty \).

For a proof, see, for example, theorem 5.2 of [35].

(Note that part 1 of the theorem requires only irreducibility, whereas part 2 requires both irreducibility and aperiodicity)
A stochastic matrix satisfying the conditions of the theorem is sometimes called uniformly ergodic.

One easy sufficient condition for aperiodicity and irreducibility is that every element of $P$ is strictly positive

- Try to convince yourself of this

### 19.7.1 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 $\alpha \to 0$.

### 19.7.2 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

```julia
In [18]: P = [.4 .6; .2 .8];
mc = MarkovChain(P);
stationary_distributions(mc)
```

Out[18]: 1-element Array{Array{Float64,1},1}:

```
[0.25, 0.7499999999999999]
```
The stationary distribution is unique.

### 19.7.3 Convergence to Stationarity

Part 2 of the Markov chain convergence theorem stated above tells us that the distribution of $X_t$ converges to the stationary distribution regardless of where we start off.

This adds considerable weight to our interpretation of $\psi^*$ as a stochastic steady state.

The convergence in the theorem is illustrated in the next figure.

```python
In [19]: P = [[0.971, 0.029, 0.000],
       [0.145, 0.778, 0.077],
       [0.000, 0.508, 0.492]] # stochastic matrix

ψ = [0.0, 0.2, 0.8] # initial distribution

t = 20 # path length
x_vals = zeros(t)
y_vals = similar(x_vals)
z_vals = similar(x_vals)
colors = [repeat([red], 20); black] # for plotting

for i in 1:t
    x_vals[i] = ψ[1]
y_vals[i] = ψ[2]
z_vals[i] = ψ[3]
    ψ = ψ * P # update distribution
end

mc = MarkovChain(P)
ψ_star = stationary_distributions(mc)[1]
x_star, y_star, z_star = ψ_star # unpack the stationary dist
plt = scatter([x_vals; x_star], [y_vals; y_star], [z_vals; z_star], color=colors,
              gridalpha = 0.5, legend = :none)
plot!(plt, camera = (45,45))
```

Out[19]:
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.

### 19.8 Ergodicity

Under irreducibility, yet another important result obtains: For all $x \in S$,

$$\frac{1}{m} \sum_{t=1}^{m} 1\{X_t = x\} \to \psi^*(x) \quad \text{as} \quad m \to \infty$$

(7)

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 (7) is valid.

The convergence in (7) is a special case of a law of large numbers result for Markov chains — see EDTC, section 4.3.4 for some additional information.
19.8.1 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.

19.9 Computing Expectations

We are interested in computing expectations of the form

$$\mathbb{E}[h(X_t)] \quad (8)$$

and conditional expectations such as

$$\mathbb{E}[h(X_{t+k}) \mid X_t = x] \quad (9)$$

where

- $\{X_t\}$ is a Markov chain generated by $n \times n$ stochastic matrix $P$
- $h$ is a given function, which, in expressions involving matrix algebra, we’ll think of as the column vector

$$h = \begin{pmatrix} h(x_1) \\ \vdots \\ h(x_n) \end{pmatrix}$$

The unconditional expectation (8) 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 (9), 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

\[
\mathbb{E}[h(X_{t+k}) \mid X_t = x] = (P^k h)(x)
\] (10)

The vector \( P^k h \) stores the conditional expectation \( \mathbb{E}[h(X_{t+k}) \mid X_t = x] \) over all \( x \).

### 19.9.1 Expectations of Geometric Sums

Sometimes we also want to compute expectations of a geometric sum, such as \( \sum \beta^t h(X_t) \).

In view of the preceding discussion, this is

\[
\mathbb{E} \left[ \sum_{j=0}^{\infty} \beta^j h(X_{t+j}) \mid X_t = x \right] = [(I - \beta P)^{-1} h](x)
\]

where

\[
(I - \beta P)^{-1} = I + \beta P + \beta^2 P^2 + \cdots
\]

Premultiplication by \( (I - \beta P)^{-1} \) amounts to “applying the resolvent operator”.

### 19.10 Exercises

#### 19.10.1 Exercise 1

According to the discussion above, if a worker’s employment dynamics obey the stochastic matrix

\[
P = \begin{pmatrix}
1 - \alpha & \alpha \\
\beta & 1 - \beta
\end{pmatrix}
\]

with \( \alpha \in (0, 1) \) and \( \beta \in (0, 1) \), then, in the long-run, the fraction of time spent unemployed will be

\[
p := \frac{\beta}{\alpha + \beta}
\]

In other words, if \( \{X_t\} \) represents the Markov chain for employment, then \( \bar{X}_m \to p \) as \( m \to \infty \), where

\[
\bar{X}_m := \frac{1}{m} \sum_{t=1}^{m} 1\{X_t = 1\}
\]

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

19.10.2 Exercise 2

A topic of interest for economics and many other disciplines is ranking.

Let's now consider one of the most practical and important ranking problems — the rank assigned to web pages by search engines.

(Although the problem is motivated from outside of economics, there is in fact a deep connection between search ranking systems and prices in certain competitive equilibria — see [25])

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 \( P(i, j) = 1\{i \to j\}/\ell_i \) where \( 1\{i \to j\} = 1 \) if \( i \) has a link to \( j \) and zero otherwise.

The matrix \( P \) is a stochastic matrix provided that each page has at least one link.
With this definition of $P$ we have

$$r_j = \sum_{i \in L_j} \ell_i = \sum_{\text{all } i} 1\{i \to j\} \ell_i = \sum_{\text{all } i} P(i, j) r_i$$

Writing $r$ for the row vector of rankings, this becomes $r = rP$.

Hence $r$ is the stationary distribution of the stochastic matrix $P$.

Let’s think of $P(i, j)$ as the probability of “moving” from page $i$ to page $j$.

The value $P(i, j)$ has the interpretation

- $P(i, j) = 1/k$ if $i$ has $k$ outbound links, and $j$ is one of them
- $P(i, j) = 0$ if $i$ has no direct link to $j$

Thus, motion from page to page is that of a web surfer who moves from one page to another by randomly clicking on one of the links on that page.

Here “random” means that each link is selected with equal probability.

Since $r$ is the stationary distribution of $P$, assuming that the uniform ergodicity condition is valid, we can interpret $r_j$ as the fraction of time that a (very persistent) random surfer spends at page $j$.

Your exercise is to apply this ranking algorithm to the graph pictured above, and return the list of pages ordered by rank.

When you solve for the ranking, you will find that the highest ranked node is in fact $g$, while the lowest is $a$.

19.10.3 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 i.i.d. and $N(0, \sigma_u^2)$.

The variance of the stationary probability distribution of $\{y_t\}$ is

$$\sigma_y^2 := \frac{\sigma_u^2}{1 - \rho^2}$$

Tauchen’s method [104] 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 let’s write our own version as an exercise.

As a first step we choose

- $n$, the number of states for the discrete approximation
• $m$, an integer that parameterizes the width of the state space

Next we create a state space $\{x_0, \ldots, x_{n-1}\} \subset \mathbb{R}$ and a stochastic $n \times n$ matrix $P$ such that

• $x_0 = -m \sigma_y$
• $x_{n-1} = m \sigma_y$
• $x_{i+1} = x_i + s$ where $s = (x_{n-1} - x_0)/(n - 1)$

Let $F$ be the cumulative distribution function of the normal distribution $N(0, \sigma_u^2)$.

The values $P(x_i, x_j)$ are computed to approximate the AR(1) process — omitting the derivation, the rules are as follows:

1. If $j = 0$, then set

$$P(x_i, x_j) = P(x_i, x_0) = F(x_0 - \rho x_i + s/2)$$

1. If $j = n - 1$, then set

$$P(x_i, x_j) = P(x_i, x_{n-1}) = 1 - F(x_{n-1} - \rho x_i - s/2)$$

1. 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.jl’s MarkovChain type.

### 19.11 Solutions

#### 19.11.1 Exercise 1

Compute the fraction of time that the worker spends unemployed, and compare it to the stationary probability.

```python
In [20]: α = 0.1 # probability of getting hired
β = 0.1 # probability of getting fired
N = 10_000
p = β / (α + β) # steady-state probabilities
P = [1 - α  α
     β  1 - β] # stochastic matrix
mc = MarkovChain(P)
labels = ["start unemployed", "start employed"]
y_vals = Array{Vector}(undef, 2) # sample paths holder
```
for x0 in 1:2
    X = simulate_indices(mc, N; init = x0)  # generate the sample path
    \hat{X} = cumsum(X .== 1) ./ (1:N)  # compute state fraction. ./ required for precedence
    y_vals[x0] = \hat{X} .- \hat{P}  # plot divergence from steady state
end

plot(y_vals, color = [:blue :green], fillrange = 0, fillalpha = 0.1, ylims = (-0.25, 0.25), label = reshape(labels, 1, length(labels)))

Out[20]:

19.11.2 Exercise 2

In [21]: web_graph_data = sort(Dict('a' => ['d', 'f'], 'b' => ['j', 'k', 'm'], 'c' => ['c', 'g', 'j', 'm'], 'd' => ['f', 'h', 'k'], 'e' => ['d', 'h', 'l'], 'f' => ['a', 'b', 'j', 'l'], 'g' => ['b', 'j'], 'h' => ['d', 'g', 'l', 'm'], 'i' => ['g', 'h', 'n'], 'j' => ['e', 'i', 'k'], 'k' => ['n'], 'l' => ['m'], 'm' => ['g'], 'n' => ['c', 'j', 'm']))

Warning: `sort(d::Dict; args...)` is deprecated, use `sort!(OrderedDict(d); args...)` instead.
| caller = top-level scope at In[21]:1
| @ Core In[21]:1

**Out[21]:** OrderedCollections.OrderedDict{Char,Array{Char,1}} with 14 entries:
- 'a' => ['d', 'f']
- 'b' => ['j', 'k', 'm']
- 'c' => ['c', 'g', 'j', 'm']
- 'd' => ['f', 'h', 'k']
- 'e' => ['d', 'h', 'l']
- 'f' => ['a', 'b', 'j', 'l']
- 'g' => ['b', 'j']
- 'h' => ['d', 'g', 'l', 'm']
- 'i' => ['g', 'h', 'n']
- 'j' => ['e', 'i', 'k']
- 'k' => ['n']
- 'l' => ['m']
- 'm' => ['g']
- 'n' => ['c', 'j', 'm']

**In [22]:**
```python
nodes = keys(web_graph_data)
n = length(nodes)
# create adjacency matrix of links (Q[i, j] = true for link, false otherwise)
Q = fill(false, n, n)
for (node, edges) in enumerate(values(web_graph_data))
    Q[node, nodes .∈ Ref(edges)] .= true
end
# create the corresponding stochastic matrix
P = Q ./ sum(Q, dims = 2)
mc = MarkovChain(P)
r = stationary_distributions(mc)[1] # stationary distribution
ranked_pages = Dict(zip(keys(web_graph_data), r)) # results holder

# print solution
println("Rankings\n***")
sort(collect(ranked_pages), by = x -> x[2], rev = true) # print sorted

***
```

**Out[22]:** 14-element Array{Pair{Char,Float64},1}:
- 'g' => 0.16070778858515053
- 'j' => 0.15936158342833578
- 'm' => 0.119515123584059
- 'n' => 0.10876973827831275
- 'k' => 0.0910628956751643
- 'b' => 0.0832646081451476
- 'e' => 0.05312052780944526
- 'i' => 0.05312052780944526
- 'c' => 0.04834210590147233
- 'h' => 0.04560118369030004
- 'l' => 0.032017852378295776
- 'd' => 0.030562495452009602
19.11.3 Exercise 3

A solution from QuantEcon.jl can be found here.

Footnotes

[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.
Chapter 20

Continuous State Markov Chains

20.1 Contents

- Overview 20.2
- The Density Case 20.3
- Beyond Densities 20.4
- Stability 20.5
- Exercises 20.6
- Solutions 20.7
- Appendix 20.8

20.2 Overview

In a previous lecture we learned about finite Markov chains, a relatively elementary class of stochastic dynamic models.

The present lecture extends this analysis to continuous (i.e., uncountable) state Markov chains.

Most stochastic dynamic models studied by economists either fit directly into this class or can be represented as continuous state Markov chains after minor modifications.

In this lecture, our focus will be on continuous Markov models that

- evolve in discrete time
- are often nonlinear

The fact that we accommodate nonlinear models here is significant, because linear stochastic models have their own highly developed tool set, as we’ll see later on.

The question that interests us most is: Given a particular stochastic dynamic model, how will the state of the system evolve over time?

In particular,

- What happens to the distribution of the state variables?
- Is there anything we can say about the “average behavior” of these variables?
- Is there a notion of “steady state” or “long run equilibrium” that’s applicable to the model?
  - If so, how can we compute it?
Answering these questions will lead us to revisit many of the topics that occupied us in the finite state case, such as simulation, distribution dynamics, stability, ergodicity, etc.

**Note**

For some people, the term “Markov chain” always refers to a process with a finite or discrete state space. We follow the mainstream mathematical literature (e.g., [77]) in using the term to refer to any discrete time Markov process.

### 20.2.1 Setup

In [1]: `using InstantiateFromURL`  
   
   # optionally add arguments to force installation: instantiate = true,  
   
   precompile = true  
   
   `github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")`

In [2]: `using LinearAlgebra, Statistics`  
   `using KernelDensity, Distributions, Plots, QuantEcon, Random`

### 20.3 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.

### 20.3.1 Definitions and Basic Properties

In our lecture on finite Markov chains, we studied discrete time Markov chains that evolve on a finite state space $S$.

In this setting, the dynamics of the model are described by a stochastic matrix — a nonnegative square matrix $P = P[i, j]$ such that each row $P[i, \cdot]$ sums to one.

The interpretation of $P$ is that $P[i, j]$ represents the probability of transitioning from state $i$ to state $j$ in one unit of time.

In symbols,

$$P\{X_{t+1} = j \mid X_t = i\} = P[i, j]$$

Equivalently,

- $P$ can be thought of as a family of distributions $P[i, \cdot]$, one for each $i \in S$
20.3. THE DENSITY CASE

- $P[i, \cdot]$ is the distribution of $X_{t+1}$ given $X_t = i$

(As you probably recall, when using Julia arrays, $P[i, \cdot]$ is expressed as $P[i, :]$)

In this section, we’ll allow $S$ to be a subset of $\mathbb{R}$, such as

- $\mathbb{R}$ itself
- the positive reals $(0, \infty)$
- a bounded interval $(a, b)$

The family of discrete distributions $P[i, \cdot]$ will be replaced by a family of densities $p(x, \cdot)$, one for each $x \in S$.

Analogous to the finite state case, $p(x, \cdot)$ is to be understood as the distribution (density) of $X_{t+1}$ given $X_t = x$.

More formally, a stochastic kernel on $S$ is a function $p: S \times S \to \mathbb{R}$ with the property that

1. $p(x, y) \geq 0$ for all $x, y \in S$
2. $\int p(x, y) dy = 1$ for all $x \in S$

(Integrals are over the whole space unless otherwise specified)

For example, let $S = \mathbb{R}$ and consider the particular stochastic kernel $p_w$ defined by

$$p_w(x, y) := \frac{1}{\sqrt{2\pi}} \exp \left\{ -\frac{(y - x)^2}{2} \right\} \quad (1)$$

What kind of model does $p_w$ represent?

The answer is, the (normally distributed) random walk

$$X_{t+1} = X_t + \xi_{t+1} \quad \text{where} \quad \{\xi_t\} \overset{\text{IID}}{\sim} N(0, 1) \quad (2)$$

To see this, let’s find the stochastic kernel $p$ corresponding to (2).

Recall that $p(x, \cdot)$ represents the distribution of $X_{t+1}$ given $X_t = x$.

Letting $X_t = x$ in (2) and considering the distribution of $X_{t+1}$, we see that $p(x, \cdot) = N(x, 1)$.

In other words, $p$ is exactly $p_w$, as defined in (1).

20.3.2 Connection to Stochastic Difference Equations

In the previous section, we made the connection between stochastic difference equation (2) and stochastic kernel (1).

In economics and time series analysis we meet stochastic difference equations of all different shapes and sizes.

It will be useful for us if we have some systematic methods for converting stochastic difference equations into stochastic kernels.

To this end, consider the generic (scalar) stochastic difference equation given by

$$X_{t+1} = \mu(X_t) + \sigma(X_t) \xi_{t+1} \quad (3)$$
Here we assume that
- \( \{ \xi_t \} \overset{\text{IID}}{\sim} \phi \), where \( \phi \) is a given density on \( \mathbb{R} \)
- \( \mu \) and \( \sigma \) are given functions on \( S \), with \( \sigma(x) > 0 \) for all \( x \)

**Example 1:** The random walk (2) is a special case of (3), with \( \mu(x) = x \) and \( \sigma(x) = 1 \).

**Example 2:** Consider the ARCH model

\[
X_{t+1} = \alpha X_t + \sigma_t \xi_{t+1}, \quad \sigma_t^2 = \beta + \gamma X_t^2, \quad \beta, \gamma > 0
\]

Alternatively, we can write the model as

\[
X_{t+1} = \alpha X_t + (\beta + \gamma X_t^2)^{1/2} \xi_{t+1}
\]

This is a special case of (3) with \( \mu(x) = \alpha x \) and \( \sigma(x) = (\beta + \gamma x^2)^{1/2} \).

**Example 3:** With stochastic production and a constant savings rate, the one-sector neoclassical growth model leads to a law of motion for capital per worker such as

\[
k_{t+1} = s A_{t+1} f(k_t) + (1 - \delta) k_t
\]

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 [68], section 3.1.2), although we omit the details here)

Equation (5) is a special case of (3) with \( \mu(x) = (1 - \delta)x \) and \( \sigma(x) = s f(x) \).

Now let’s obtain the stochastic kernel corresponding to the generic model (3).

To find it, note first that if \( U \) is a random variable with density \( f_U \), and \( V = a + bU \) for some constants \( a, b \) with \( b > 0 \), then the density of \( V \) is given by

\[
f_V(v) = \frac{1}{b} f_U \left( \frac{v - a}{b} \right)
\]

(The proof is below. For a multidimensional version see EDTC, theorem 8.1.3)

Taking (6) as given for the moment, we can obtain the stochastic kernel \( p \) for (3) by recalling that \( p(x, \cdot) \) is the conditional density of \( X_{t+1} \) given \( X_t = x \).

In the present case, this is equivalent to stating that \( p(x, \cdot) \) is the density of \( Y := \mu(x) + \sigma(x) \xi_{t+1} \) when \( \xi_{t+1} \sim \phi \).

Hence, by (6),

\[
p(x, y) = \frac{1}{\sigma(x)} \phi \left( \frac{y - \mu(x)}{\sigma(x)} \right)
\]

For example, the growth model in (5) has stochastic kernel
20.3. THE DENSITY CASE

\[ p(x, y) = \frac{1}{sf(x)} \phi \left( \frac{y - (1 - \delta)x}{sf(x)} \right) \]  

(8)

where \( \phi \) is the density of \( A_{t+1} \).

(Regarding the state space \( S \) for this model, a natural choice is \((0, \infty)\) — in which case \( \sigma(x) = sf(x) \) is strictly positive for all \( s \) as required)

20.3.3 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

\[ \psi_{t+1}(y) = \int p(x, y) \psi_t(x) dx, \quad \forall y \in S \]  

(9)

It is convenient to think of this updating process in terms of an operator.

(An operator is just a function, but the term is usually reserved for a function that sends functions into functions)

Let \( D \) be the set of all densities on \( S \), and let \( P \) be the operator from \( D \) to itself that takes density \( \psi \) and sends it into new density \( \psi P \), where the latter is defined by

\[ (\psi P)(y) = \int p(x, y) \psi(x) dx \]  

(10)

This operator is usually called the Markov operator corresponding to \( p \)

**Note**

Unlike most operators, we write \( P \) to the right of its argument, instead of to the left (i.e., \( \psi P \) instead of \( P \psi \)). This is a common convention, with the intention being to maintain the parallel with the finite case — see here.

With this notation, we can write (9) more succinctly as \( \psi_{t+1}(y) = (\psi_t P)(y) \) for all \( y \), or, dropping the \( y \) and letting “=” indicate equality of functions,
\[ \psi_{t+1} = \psi_t P \]  

Equation (11) tells us that if we specify a distribution for \( \psi_0 \), then the entire sequence of future distributions can be obtained by iterating with \( P \).

It’s interesting to note that (11) is a deterministic difference equation.

Thus, by converting a stochastic difference equation such as (3) into a stochastic kernel \( p \) and hence an operator \( P \), we convert a stochastic difference equation into a deterministic one (albeit in a much higher dimensional space).

**Note**

Some people might be aware that discrete Markov chains are in fact a special case of the continuous Markov chains we have just described. The reason is that probability mass functions are densities with respect to the **counting measure**.

### 20.3.4 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 (10) and (11) using numerical integration.

However, to produce \( \psi P \) from \( \psi \) via (10), you would need to integrate at every \( y \), and there is a continuum of such \( y \).

Another possibility is to discretize the model, but this introduces errors of unknown size.

A nicer alternative in the present setting is to combine simulation with an elegant estimator called the **look ahead** estimator.

Let’s go over the ideas with reference to the growth model **discussed above**, the dynamics of which we repeat here for convenience:

\[ k_{t+1} = sA_{t+1}f(k_t) + (1 - \delta)k_t \]  

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 (12)

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 $k_{t-1}^1, \ldots, k_{t-1}^n$ and form the estimate

$$\psi^n_t(y) = \frac{1}{n} \sum_{i=1}^{n} p(k_{t-1}^i, y)$$

where $p$ is the growth model stochastic kernel in (8).

What is the justification for this slightly surprising estimator?

The idea is that, by the strong law of large numbers,

$$\frac{1}{n} \sum_{i=1}^{n} p(k_{t-1}^i, y) \to \mathbb{E}p(k_{t-1}^i, y) = \int p(x, y) \psi_{t-1}(x) \, dx = \psi_t(y)$$

with probability one as $n \to \infty$.

Here the first equality is by the definition of $\psi_{t-1}$, and the second is by (9).

We have just shown that our estimator $\psi^n_t(y)$ in (13) converges almost surely to $\psi_t(y)$, which is just what we want to compute.

In fact much stronger convergence results are true (see, for example, this paper).

### 20.3.5 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 (13) using

- an object of type LAE that stores the stochastic kernel and the observations
- the value $y$ as its second argument

The function is vectorized, in the sense that if `psi` is such an instance and `y` is an array, then the call `psi(y)` acts elementwise.

(This is the reason that we reshaped $X$ and $y$ inside the type — to make vectorization work)

### 20.3.6 Example

The following code is example of usage for the stochastic growth model described above

```julia
using Distributions, StatsPlots, Plots, QuantEcon, Random
gr(fmt = :png)
Random.seed!(42) # For deterministic results.

s = 0.2
```
\[ \delta = 0.1 \]
\[ a_\sigma = 0.4 \]  # A = exp(B) where B \sim N(\theta, a_\sigma)
\[ \alpha = 0.4 \]  # We set \( f(k) = k^{**\alpha} \)
\[ \psi_0 = \text{Beta}(5.0, 5.0) \]  # Initial distribution
\[ \phi = \text{LogNormal}(0.0, a_\sigma) \]

```python
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.\alpha
    
    pdf_arg = clamp.((y .- (1-\delta) .* x) .\d, eps(), Inf)
    return pdf.(\phi, pdf_arg) .\d
end
```

```python
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 = zeros(n, T)
\A = rand!(\phi, zeros(n, T))

# Draw first column from initial distribution
\k[:, 1] = rand(\psi_0, n) .\2  # divide by 2 to match scale = 0.5 in py version
for \t in 1:T-1
    \k[:, t+1] = s*\A[:, t] .\* \k[:, t].\alpha + (1-\delta) .\* \k[:, t]
end

# Generate \( T \) instances of LAE using this data, one for each date \( t \)
\laes = [LAE(p, \k[:, t]) for \t in T:-1:1]

# Plot
\ygrid = range(0.01, 4, length = 200)
\laes_plot = []
\colors = []
for \i in 1:T
    \psi = laes[\i]
    push!(\laes_plot, lae_est(\psi, \ygrid))
    push!(\colors, RGBA(0, 0, 0, 1 - (\i - 1)/T))
end
plot(\ygrid, \laes_plot, color = reshape(\colors, 1, length(\colors)), lw = 2, xlab = "capital", legend = :none)
t = "Density of k_1 (lighter) to k_T (darker) for T=$T"
pplot!(title = t)
```

Out[3]:
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).

### 20.4 Beyond Densities

Up until now, we have focused exclusively on continuous state Markov chains where all conditional distributions \( p(\cdot, \cdot) \) 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.

### 20.4.1 Example and Definitions

To illustrate the issues, recall that Hopenhayn and Rogerson [55] study a model of firm dynamics where individual firm productivity follows the exogenous process

\[
X_{t+1} = a + \rho X_t + \xi_{t+1}, \quad \text{where} \quad \{\xi_t\} \overset{\text{iid}}{\sim} N(0, \sigma^2)
\]
As is, this fits into the density case we treated above.

However, the authors wanted this process to take values in $[0, 1]$, so they added boundaries at the end points 0 and 1.

One way to write this is

$$X_{t+1} = h(a + \rho X_t + \xi_{t+1}) \quad \text{where} \quad h(x) := x \mathbf{1}_{\{0 \leq x \leq 1\}} + \mathbf{1}_{\{x > 1\}}$$

If you think about it, you will see that for any given $x \in [0, 1]$, the conditional distribution of $X_{t+1}$ given $X_t = x$ puts positive probability mass on 0 and 1.

Hence it cannot be represented as a density.

What we can do instead is use cumulative distribution functions (cdfs).

To this end, set

$$G(x, y) := \mathbb{P}\{h(a + \rho x + \xi_{t+1}) \leq y\} \quad (0 \leq x, y \leq 1)$$

This family of cdfs $G(x, \cdot)$ plays a role analogous to the stochastic kernel in the density case.

The distribution dynamics in (9) are then replaced by

$$F_{t+1}(y) = \int G(x, y) F_t(dx) \quad (14)$$

Here $F_t$ and $F_{t+1}$ are cdfs representing the distribution of the current state and next period state.

The intuition behind (14) is essentially the same as for (9).

### 20.4.2 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.

### 20.5 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.
20.5. STABILITY

20.5.1 Theoretical Results

Analogous to the finite case, given a stochastic kernel $p$ and corresponding Markov operator as defined in (10), a density $\psi^*$ on $S$ is called stationary for $P$ if it is a fixed point of the operator $P$.

In other words,

$$\psi^*(y) = \int p(x, y)\psi^*(x) \, dx, \quad \forall y \in S$$  \hspace{1cm} (15)

As with the finite case, if $\psi^*$ is stationary for $P$, and the distribution of $X_0$ is $\psi^*$, then, in view of (11), $X_t$ will have this same distribution for all $t$.

Hence $\psi^*$ is the stochastic equivalent of a steady state.

In the finite case, we learned that at least one stationary distribution exists, although there may be many.

When the state space is infinite, the situation is more complicated.

Even existence can fail very easily.

For example, the random walk model has no stationary density (see, e.g., EDTC, p. 210).

However, there are well-known conditions under which a stationary density $\psi^*$ exists.

With additional conditions, we can also get a unique stationary density ($\psi \in \mathcal{D}$ and $\psi = \psi P \implies \psi = \psi^*$), and also global convergence in the sense that

$$\forall \psi \in \mathcal{D}, \quad \psi P^t \rightarrow \psi^* \quad \text{as} \quad t \rightarrow \infty$$ \hspace{1cm} (16)

This combination of existence, uniqueness and global convergence in the sense of (16) is often referred to as global stability.

Under very similar conditions, we get ergodicity, which means that

$$\frac{1}{n} \sum_{t=1}^{n} h(X_t) \rightarrow \int h(x)\psi^*(x) \, dx \quad \text{as} \quad n \rightarrow \infty$$ \hspace{1cm} (17)

for any (measurable) function $h: S \rightarrow \mathbb{R}$ such that the right-hand side is finite.

Note that the convergence in (17) does not depend on the distribution (or value) of $X_0$.

This is actually very important for simulation — it means we can learn about $\psi^*$ (i.e., approximate the right hand side of (17) via the left hand side) without requiring any special knowledge about what to do with $X_0$.

So what are these conditions we require to get global stability and ergodicity?

In essence, it must be the case that

1. Probability mass does not drift off to the “edges” of the state space
2. Sufficient “mixing” obtains

For one such set of conditions see theorem 8.2.14 of EDTC.

In addition
• [100] contains a classic (but slightly outdated) treatment of these topics.
• From the mathematical literature, [64] and [77] 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.

20.5.2 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 (16) — by simulating the path of densities from various initial conditions.

Here is such a figure

All sequences are converging towards the same limit, regardless of their initial condition.

The details regarding initial conditions and so on are given in this exercise, where you are asked to replicate the figure.

20.5.3 Computing Stationary Densities

In the preceding figure, each sequence of densities is converging towards the unique stationary density $\psi^*$. Even from this figure we can get a fair idea what $\psi^*$ looks like, and where its mass is located.
However, there is a much more direct way to estimate the stationary density, and it involves only a slight modification of the look ahead estimator.

Let’s say that we have a model of the form (3) that is stable and ergodic.

Let \( p \) be the corresponding stochastic kernel, as given in (7).

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

\[
\psi^*_n(y) = \frac{1}{n} \sum_{t=1}^{n} p(X_t, y) \tag{18}
\]

This is essentially the same as the look ahead estimator (13), except that now the observations we generate are a single time series, rather than a cross section.

The justification for (18) 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 (17) and the equality on the right is by (15).

The right hand side is exactly what we want to compute.

On top of this asymptotic result, it turns out that the rate of convergence for the look ahead estimator is very good.

The first exercise helps illustrate this point.

20.6 Exercises

20.6.1 Exercise 1

Consider the simple threshold autoregressive model

\[
X_{t+1} = \theta |X_t| + (1 - \theta^2)^{1/2} \xi_{t+1} \quad \text{where} \quad \{\xi_t\} \sim \text{IID} \sim N(0, 1) \tag{19}
\]

This is one of those rare nonlinear stochastic models where an analytical expression for the stationary density is available.

In particular, provided that \( |\theta| < 1 \), there is a unique stationary density \( \psi^* \) given by

\[
\psi^*(y) = 2 \phi(y) \Phi \left[ \frac{\theta y}{(1 - \theta^2)^{1/2}} \right] \tag{20}
\]

Here \( \phi \) is the standard normal density and \( \Phi \) is the standard normal cdf.

As an exercise, compute the look ahead estimate of \( \psi^* \), as defined in (18), and compare it with \( \psi^* \) in (20) to see whether they are indeed close for large \( n \).

In doing so, set \( \theta = 0.8 \) and \( n = 500 \).

The next figure shows the result of such a computation.
The additional density (black line) is a nonparametric kernel density estimate, added to the solution for illustration.

(You can try to replicate it before looking at the solution if you want to)

As you can see, the look ahead estimator is a much tighter fit than the kernel density estimator.

If you repeat the simulation you will see that this is consistently the case.

20.6.2 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

\[
\psi_0 = \text{Beta}(5.0, 5.0) \quad \# \text{Initial distribution}
\]

\[n = 1000\]

\[
\# \quad \# \quad \# \quad \text{.... more setup}
\]

\[
\text{for } i \text{ in } 1:4
\]

\[
\quad \# \quad \# \quad \# \quad \text{.... some code}
\]

\[
\quad \text{rand_draws} = (\text{rand}(\psi_0, n) + 2.5i) \div 2
\]
20.6.3 Exercise 3

A common way to compare distributions visually is with boxplots.

To illustrate, let’s generate three artificial data sets and compare them with a boxplot

```python
In [4]: 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 = ["X", "Y", "Z"]
xlabels = reshape(repeat(l, n), 3n, 1)
boxplot(xlabels, data, label = "", ylims = (-2, 14))

Out[4]:
```

The three data sets are

\[
\{X_1, \ldots, X_n\} \sim LN(0,1), \quad \{Y_1, \ldots, Y_n\} \sim N(2,1), \quad \text{and} \quad \{Z_1, \ldots, Z_n\} \sim N(4,1),
\]

The figure looks as follows.

Each data set is represented by a box, where the top and bottom of the box are the third and first quartiles of the data, and the red line in the center is the median.

The boxes give some indication as to

- the location of probability mass for each sample
- whether the distribution is right-skewed (as is the lognormal distribution), etc
Now let’s put these ideas to use in a simulation.

Consider the threshold autoregressive model in (19).

We know that the distribution of $X_t$ will converge to (20) whenever $|\theta| < 1$.

Let’s observe this convergence from different initial conditions using boxplots.

In particular, the exercise is to generate $J$ boxplot figures, one for each initial condition $X_0$ in

\[
\text{initial\_conditions} = \text{range}(8, 0, \text{length} = J)
\]

For each $X_0$ in this set,

1. Generate $k$ time series of length $n$, each starting at $X_0$ and obeying (19).
2. Create a boxplot representing $n$ distributions, where the $t$-th distribution shows the $k$ observations of $X_t$.

Use $\theta = 0.9, n = 20, k = 5000, J = 8$.

## 20.7 Solutions

In [5]: `using` KernelDensity

### 20.7.1 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.

In [6]: `ϕ = Normal()`
`n = 500`
`θ = 0.8`
`d = sqrt(1.0 - θ^2)`
`δ = θ / d`

# true density of TAR model
`ψ_*=(y) = 2 .* pdf.(ϕ, y) .* cdf.(ϕ, δ * y)`

# Stochastic kernel for the TAR model.
`p_TAR(x, y) = pdf.(ϕ, (y - δ * abs.(x)) ./ d) ./ d`

`Z = rand(ϕ, n)`
`X = zeros(n)`
`for t in 1:n-1`
`    X[t+1] = θ * abs(X[t]) + d * Z[t]`
`end`
\[ \psi_{\text{est}}(a) = \text{lae}_{\text{est}}(\text{LAE}(p_{\text{TAR}}, X), a) \]
\[ k_{\text{est}} = \text{kde}(X) \]

\[ y_s = \text{range}(-3, 3, \text{length} = 200) \]
plot(ys, \( \psi_{\text{star}}(ys) \), color=:blue, lw = 2, alpha = 0.6, label = "true")
plot!(ys, \( \psi_{\text{est}}(ys) \), 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")

Out[6]:

20.7.2 Exercise 2

Here’s one program that does the job.

In [7]:
\[
\begin{align*}
    s &= 0.2 \\
    \delta &= 0.1 \\
    a_\sigma &= 0.4 \quad \# A = \exp(B) \text{ where } B \sim N(0, a_\sigma) \\
    \alpha &= 0.4 \quad \# \text{We set } f(k) = k^{\alpha} \\
    \psi_{\theta} &= \text{Beta}(5.0, 5.0) \quad \# \text{Initial distribution} \\
    \phi &= \text{LogNormal}(0.0, a_\sigma)
\end{align*}
\]

function p_growth(x, y)
    # Stochastic kernel for the growth model with Cobb-Douglas production.
    # Both x and y must be strictly positive.
    \[ d = s * x^{\alpha} \]
    pdf_arg = clamp(((y - (1-\delta) .* x) ./ d, eps(), Inf)
return pdf.(\( \phi \), pdf_arg) ./ d
CHAPTER 20. CONTINUOUS STATE MARKOV CHAINS

n = 1000  # Number of observations at each date t
T = 40    # Compute density of k_t at 1,...,T+1

xmax = 6.5
ygrid = range(0.01, xmax, length = 150)
laes_plot = zeros(length(ygrid), 4T)
colors = []
for i in 1:4
    k = zeros(n, T)
    A = rand!(ϕ, zeros(n, T))

    # Draw first column from initial distribution
    # match scale = 0.5 and loc = 2i in julia version
    k[:, 1] = (rand(ψ_0, n) .+ 2.5i) ./ 2
    for t in 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 in T:-1:1]
    ind = i
    for j in 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)
colors = reshape(colors, 1, length(colors))
plot(ygrid, laes_plot, layout = (2,2), color = colors,
     legend = :none, xlabel = "capital", xlims = (0, xmax))
20.7.3 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.

In [8]:

```python
n = 20
k = 5000
J = 6
θ = 0.9
d = sqrt(1 - θ^2)
δ = θ / d

initial_conditions = range(θ, θ, length = J)

Z = randn(k, n, J)
titles = []
data = []
x_labels = []

for j in 1:J
    title = "time series from t = \$(initial_conditions[j])\"
push!(titles, title)

    X = zeros(k, n)
    X[:, 1] .= initial_conditions[j]
    labels = []
    labels = vcat(labels, ones(k, 1))

    for t in 2:n
        X[:, t] = θ .* abs.(X[:, t-1]) + d .* Z[:, t, j]
        labels = vcat(labels, t^ones(k, 1))
    end

X = reshape(X, n*k, 1)
```

In [9]: plots = []
for i in 1:J
    push!(plots, boxplot(vec(x_labels[i]), vec(data[i]), title = titles[i]))
end
plot(plots..., layout = (J, 1), legend = :none, size = (800, 2000))
20.8 Appendix

Here's the proof of (6).

Let $F_U$ and $F_V$ be the cumulative distributions of $U$ and $V$ respectively.

By the definition of $V$, we have $F_V(v) = \mathbb{P}\{a + bU \leq v\} = \mathbb{P}\{U \leq (v - a)/b\}$.

In other words, $F_V(v) = F_U((v - a)/b)$.

Differentiating with respect to $v$ yields (6).
Chapter 21

A First Look at the Kalman Filter

21.1 Contents

• Overview 21.2
• The Basic Idea 21.3
• Convergence 21.4
• Implementation 21.5
• Exercises 21.6
• Solutions 21.7

21.2 Overview

This lecture provides a simple and intuitive introduction to the Kalman filter, for those who either

• have heard of the Kalman filter but don’t know how it works, or
• know the Kalman filter equations, but don’t know where they come from

For additional (more advanced) reading on the Kalman filter, see

• [68], section 2.7.
• [3]

The second reference presents a comprehensive treatment of the Kalman filter.

Required knowledge: Familiarity with matrix manipulations, multivariate normal distributions, covariance matrices, etc.

21.2.1 Setup

In [1]: using InstantiateFromURL
   # optionally add arguments to force installation: instantiate = true,
   # precompile = true
   github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

In [2]: using LinearAlgebra, Statistics
21.3 The Basic Idea

The Kalman filter has many applications in economics, but for now let’s pretend that we are rocket scientists.

A missile has been launched from country Y and our mission is to track it.

Let $x \in \mathbb{R}^2$ denote the current location of the missile—a pair indicating latitude-longitude coordinates on a map.

At the present moment in time, the precise location $x$ is unknown, but we do have some beliefs about $x$.

One way to summarize our knowledge is a point prediction $\hat{x}$

- But what if the President wants to know the probability that the missile is currently over the Sea of Japan?
- Then it is better to summarize our initial beliefs with a bivariate probability density $p$.
  - $\int_E p(x) dx$ indicates the probability that we attach to the missile being in region $E$

The density $p$ is called our prior for the random variable $x$.

To keep things tractable in our example, we assume that our prior is Gaussian. In particular, we take

$$ p = N(\hat{x}, \Sigma) $$

(1)

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

$$ \hat{x} = \begin{pmatrix} 0.2 \\ -0.2 \end{pmatrix}, \quad \Sigma = \begin{pmatrix} 0.4 & 0.3 \\ 0.3 & 0.45 \end{pmatrix} $$

(2)

This density $p(x)$ is shown below as a contour map, with the center of the red ellipse being equal to $\hat{x}$

In [3]: using Plots, Distributions
gr(fmt = :png); # plots setup

In [4]: # set up prior objects
    Σ = [0.4 0.3
         0.3 0.45]
    ̂x = [0.2, -0.2]

    # define G and R from the equation $y = Gx + N(0, R)$
    G = I # this is a generic identity object that conforms to the right dimensions
    R = 0.5 .* Σ

    # define A and Q
    A = [1.2 0
         0 -0.2]
    Q = 0.3Σ

    y = [2.3, -1.9]
21.3. THE BASIC IDEA

21.3.1 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 \( y \).

In [5]: # plot the figure
    annotate!(y[1], y[2], "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

$$y = Gx + v, \quad \text{where} \quad v \sim N(0, R)$$

(3)

Here $G$ and $R$ are $2 \times 2$ matrices with $R$ positive definite. Both are assumed known, and the noise term $v$ is assumed to be independent of $x$.

How then should we combine our prior $p(x) = N(\hat{x}, \Sigma)$ and this new information $y$ to improve our understanding of the location of the missile?

As you may have guessed, the answer is to use Bayes’ theorem, which tells us to update our prior $p(x)$ to $p(x | y)$ via

$$p(x | y) = \frac{p(y | x) p(x)}{p(y)}$$

where $p(y) = \int p(y | x) p(x) dx$.

In solving for $p(x | y)$, we observe that

- $p(x) = N(\hat{x}, \Sigma)$
- In view of (3), the conditional density $p(y | x)$ is $N(Gx, R)$
- $p(y)$ does not depend on $x$, and enters into the calculations only as a normalizing constant

Because we are in a linear and Gaussian framework, the updated density can be computed by calculating population linear regressions.

In particular, the solution is known Section ?? to be

$$p(x | y) = N(\hat{x}^F, \Sigma^F)$$
21.3. THE BASIC IDEA

\[ \hat{x}^F := \hat{x} + \Sigma G'(G \Sigma G' + R)^{-1}(y - G\hat{x}) \quad \text{and} \quad \Sigma^F := \Sigma - \Sigma G'(G \Sigma G' + R)^{-1}G \Sigma \]  

(4)

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.

In [6]:

```python
# define posterior objects
M = Σ * G' * inv(G * Σ * G' + R)
x_F = x + M * (y - G * x)
Σ_F = Σ - M * G * Σ

# plot the new density on the old plot
newdist = MvNormal(x_F, Symmetric(Σ_F))  # because Σ_F
contour!(x_grid, y_grid, two_args_to_pdf(newdist), fill = false, color = :lightest, cbar = false)
contour!(x_grid, y_grid, two_args_to_pdf(newdist), fill = false, levels = 7, color = :grays, cbar = false)
contour!(x_grid, y_grid, two_args_to_pdf(dist), fill = false, levels = 7, lw = 1, color = :grays, cbar = false)
```

Out[6]:

Our new density twists the prior \( p(x) \) in a direction determined by the new information \( y - G\hat{x} \).
In generating the figure, we set $G$ to the identity matrix and $R = 0.5\Sigma$ for $\Sigma$ defined in (2).

### 21.3.2 The Forecast Step

What have we achieved so far?

We have obtained probabilities for the current location of the state (missile) given prior and current information.

This is called “filtering” rather than forecasting, because we are filtering out noise rather than looking into the future

- $p(x \mid y) = N(\hat{x}^F, \Sigma^F)$ is called the filtering distribution

But now let’s suppose that we are given another task: to predict the location of the missile after one unit of time (whatever that may be) has elapsed.

To do this we need a model of how the state evolves.

Let’s suppose that we have one, and that it’s linear and Gaussian. In particular,

$$x_{t+1} = Ax_t + w_{t+1}, \text{ where } w_t \sim N(0, Q)$$

Our aim is to combine this law of motion and our current distribution $p(x \mid y) = N(\hat{x}^F, \Sigma^F)$ to come up with a new predictive distribution for the location in one unit of time.

In view of (5), all we have to do is introduce a random vector $x^F \sim N(\hat{x}^F, \Sigma^F)$ and work out the distribution of $Ax^F + w$ where $w$ is independent of $x^F$ and has distribution $N(0, Q)$.

Since linear combinations of Gaussians are Gaussian, $Ax^F + w$ is Gaussian.

Elementary calculations and the expressions in (4) tell us that

$$\mathbb{E}[Ax^F + w] = A\mathbb{E}x^F + \mathbb{E}w = A\hat{x} + A\Sigma G'(G\Sigma G' + R)^{-1}(y - G\hat{x})$$

and

$$\text{Var}[Ax^F + w] = A \text{Var}[x^F]A' + Q = A\Sigma F A' + Q = A\Sigma A' - A\Sigma G'(G\Sigma G' + R)^{-1}G\Sigma A' + Q$$

The matrix $A\Sigma G'(G\Sigma G' + R)^{-1}$ is often written as $K_\Sigma$ and called the Kalman gain.

- The subscript $\Sigma$ has been added to remind us that $K_\Sigma$ depends on $\Sigma$, but not $y$ or $\hat{x}$.

Using this notation, we can summarize our results as follows.

Our updated prediction is the density $N(\hat{x}_{new}, \Sigma_{new})$ where

$$\hat{x}_{new} := A\hat{x} + K_\Sigma(y - G\hat{x})$$

$$\Sigma_{new} := A\Sigma A' - K_\Sigma G\Sigma A' + Q$$

- The density $p_{new}(x) = N(\hat{x}_{new}, \Sigma_{new})$ is called the predictive distribution.

The predictive distribution is the new density shown in the following figure, where the update has used parameters
21.3. THE BASIC IDEA

\[ A = \begin{pmatrix} 1.2 & 0.0 \\ 0.0 & -0.2 \end{pmatrix}, \quad Q = 0.3 \Sigma \]

In [7]: # get the predictive distribution

\[
\text{new}_x = A \times \hat{x}_F \\
\text{new}_\Sigma = A \times \Sigma_F \times A' + Q \\
predictdist = \text{MvNormal}(\text{new}_x, \text{Symmetric}(\text{new}_\Sigma))
\]

# plot Density 3

```
contour(x_grid, y_grid, two_args_to_pdf(predictdist), fill = false, lw = 1, color = :lighttest, cbar = false)
contour!(x_grid, y_grid, two_args_to_pdf(dist), color = :grays, cbar = false)
contour!(x_grid, y_grid, two_args_to_pdf(newdist), fill = false, levels = 7, color = :grays, cbar = false)
annotate!(y[1], y[2], "y", color = :black)
```

Out[7]:

21.3.3 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 (5) 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 $p_{new}(x)$, the full recursive procedure is:

1. Start the current period with prior $p_t(x) = N(\hat{x}_t, \Sigma_t)$.
2. Observe current measurement $y_t$.
3. Compute the filtering distribution $p_t(x | y) = N(\hat{x}_F, \Sigma_F)$ from $p_t(x)$ and $y_t$, applying Bayes rule and the conditional distribution (3).
4. Compute the predictive distribution $p_{t+1}(x) = N(\hat{x}_{t+1}, \Sigma_{t+1})$ from the filtering distribution and (5).
5. Increment $t$ by one and go to step 1.

Repeating (6), the dynamics for $\hat{x}_t$ and $\Sigma_t$ are as follows

$$\hat{x}_{t+1} = A\hat{x}_t + K_{\Sigma_t} (y_t - G\hat{x}_t)$$
$$\Sigma_{t+1} = A\Sigma_t A' - K_{\Sigma_t} G\Sigma_t A' + Q$$

(7)

These are the standard dynamic equations for the Kalman filter (see, for example, [68], page 58).

21.4 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 (5) 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 (7):

$$\Sigma_{t+1} = A\Sigma_t A' - A\Sigma_t G'(G\Sigma_t G' + R)^{-1}G\Sigma_t A' + Q$$

(8)

This is a nonlinear difference equation in $\Sigma_t$.

A fixed point of (8) is a constant matrix $\Sigma$ such that

$$\Sigma = A\Sigma A' - A\Sigma G'(G\Sigma G' + R)^{-1}G\Sigma A' + Q$$

(9)

Equation (8) is known as a discrete time Riccati difference equation.
Equation (9) 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 [4] and [3], chapter 4.
A sufficient (but not necessary) condition is that all the eigenvalues \( \lambda_i \) of \( A \) satisfy \( |\lambda_i| < 1 \) (cf. e.g., [3], 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 (8) converges to a nonnegative symmetric matrix \( \Sigma \) that solves (9).

### 21.5 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 \( (\hat{x}_F^t, \Sigma_F^t) \)
  - `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 (9) and the corresponding (stationary) Kalman gain

You can view the program on GitHub.

### 21.6 Exercises

#### 21.6.1 Exercise 1

Consider the following simple application of the Kalman filter, loosely based on [68], section 2.9.2.

Suppose that

- all variables are scalars
- the hidden state \( \{x_t\} \) is in fact constant, equal to some \( \theta \in \mathbb{R} \) unknown to the modeler

State dynamics are therefore given by (5) 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 [68], 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
21.6.2 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.
21.6.3 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.

**21.6.4 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 (9)) 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.

**21.7 Solutions**

In [8]: using QuantEcon

**21.7.1 Exercise 1**

In [9]: # parameters
\[ \theta = 10 \]
A, G, Q, R = 1.0, 1.0, 0.0, 1.0
\( \hat{x}_0, \Sigma_0 = 8.0, 1.0 \)

# initialize Kalman filter
kalman = Kalman(A, G, Q, R)
set_state!(kalman, \( \hat{x}_0, \Sigma_0 \))

xgrid = range(\( \theta - 5 \), \( \theta + 2 \), length = 200)
densities = zeros(200, 5) # one column per round of updating
for i in 1:5
  # record the current predicted mean and variance, and plot their densities
  m, v = kalman.cur_x_hat, kalman.cur_sigma
densities[:, i] = pdf.(Normal(m, sqrt(v)), xgrid)

  # generate the noisy signal
  y = \( \theta + \text{randn}() \)

  # update the Kalman filter
  update!(kalman, y)
end

labels = ["t=1", "t=2", "t=3", "t=4", "t=5"]
plot(xgrid, densities, label = labels, legend = :topleft, grid = false,
title = "First 5 densities when \( \theta = 10 \)"

Out[9]:

First 5 densities when \( \theta = 10 \)

21.7.2 Exercise 2

In [10]: using Random, Expectations
   Random.seed!(43) # reproducible results
\[ \epsilon = 0.1 \]
kalman = Kalman(A, G, Q, R)
set_state!(kalman, \( \mathbf{x}_0 \), \( \Sigma_0 \))
nodes, weights = qnwlege(21, \( \theta - \epsilon \), \( \theta + \epsilon \))

T = 600
z = zeros(T)
for t in 1:T
    # record the current predicted mean and variance, and plot their densities
    m, v = kalman.cur_x_hat, kalman.cur_sigma
    dist = Truncated(Normal(m, sqrt(v)), \( \theta - 30 \epsilon \), \( \theta + 30 \epsilon \)) # define on compact interval, so we can use gridded expectation
    E = expectation(dist, nodes) # nodes \( [\theta - \epsilon, \theta + \epsilon] \)
    integral = E(x -> 1) # just take the pdf integral
    z[t] = 1. - integral # generate the noisy signal and update the Kalman filter
    update!(kalman, \( \theta + \text{randn()} \))
end

plot(1:T, z, fillrange = 0, color = :blue, fillalpha = 0.2, grid = false, xlims=(0, T), legend = false)

Out[10]:

21.7.3 Exercise 3

In [11]: # define A, Q, G, R
G = I + zeros(2, 2)
R = 0.5 .* G
\begin{verbatim}
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]

# initialize the Kalman filter
kn = Kalman(A, G, Q, R)
set_state!(kn, x, Σ)

# set the true initial value of the state
x = zeros(2)

# print eigenvalues of A
println("Eigenvalues of A:
\$\{eigvals(A)\}\n")

# print stationary Σ
S, K = stationary_values(kn)
println("Stationary prediction error variance:
\$S\$")

# generate the plot
T = 50
e1 = zeros(T)
e2 = similar(e1)
for t in 1:T

    # generate signal and update prediction
    dist = MultivariateNormal(G * x, R)
y = rand(dist)
update!(kn, y)

    # update state and record error
    Ax = A * x
    x = rand(MultivariateNormal(Ax, Q))
e1[t] = sum((a - b)^2 for (a, b) in zip(x, kn.cur_x_hat))
e2[t] = sum((a - b)^2 for (a, b) in zip(x, Ax))
end

plot(1:T, e1, color = :black, linewidth = 2, alpha = 0.6, label = "Kalman\-\-filter error",
     grid = false)
plot!(1:T, e2, color = :green, linewidth = 2, alpha = 0.6,
     label = "conditional expectation error")

Eigenvalues of A:
[-0.10000000000000003, 0.8999999999999999]
Stationary prediction error variance:
[0.4032910794778669 0.10507180275061759; 0.1050718027506176 0.41061709375220456]
\end{verbatim}
Footnotes

[1] See, for example, page 93 of [14]. To get from his expressions to the ones used above, you will also need to apply the Woodbury matrix identity.
Chapter 22

Numerical Linear Algebra and Factorizations

22.1 Contents

• Overview 22.2
• Factorizations 22.3
• Continuous-Time Markov Chains (CTMCs) 22.4
• Banded Matrices 22.5
• Implementation Details and Performance 22.6
• Exercises 22.7

You cannot learn too much linear algebra. – Benedict Gross

22.2 Overview

In this lecture, we examine the structure of matrices and linear operators (e.g., dense, sparse, symmetric, tridiagonal, banded) and discuss how the structure can be exploited to radically increase the performance of solving large problems.

We build on applications discussed in previous lectures: linear algebra, orthogonal projections, and Markov chains.

The methods in this section are called direct methods, and they are qualitatively similar to performing Gaussian elimination to factor matrices and solve systems of equations. In iterative methods and sparsity we examine a different approach, using iterative algorithms, where we can think of more general linear operators.

The list of specialized packages for these tasks is enormous and growing, but some of the important organizations to look at are JuliaMatrices, JuliaSparse, and JuliaMath

NOTE: As this section uses advanced Julia techniques, you may wish to review multiple-dispatch and generic programming in introduction to types, and consider further study on generic programming.

The theme of this lecture, and numerical linear algebra in general, comes down to three principles:
1. **Identify structure** (e.g., symmetric, sparse, diagonal) matrices in order to use specialized algorithms.

2. **Do not lose structure** by applying the wrong numerical linear algebra operations at the wrong times (e.g., sparse matrix becoming dense)

3. Understand the **computational complexity** of each algorithm, given the structure of the inputs.

### 22.2.1 Setup

In [1]: `using InstantiateFromURL
   # optionally add arguments to force installation: instantiate = true, [precompile = true
   github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")`   

In [2]: `using LinearAlgebra, Statistics, BenchmarkTools, SparseArrays, Random
   Random.seed!(42); # seed random numbers for reproducibility`

### 22.2.2 Computational Complexity

Ask yourself whether the following is a **computationally expensive** operation as the matrix size increases

- Multiplying two matrices?
  - **Answer**: It depends. Multiplying two diagonal matrices is trivial.
- Solving a linear system of equations?
  - **Answer**: It depends. If the matrix is the identity, the solution is the vector itself.
- Finding the eigenvalues of a matrix?
  - **Answer**: It depends. The eigenvalues of a triangular matrix are the diagonal elements.

As the goal of this section is to move toward numerical methods with large systems, we need to understand how well algorithms scale with the size of matrices, vectors, etc. This is known as **computational complexity**. As we saw in the answer to the questions above, the algorithm - and hence the computational complexity - changes based on matrix structure.

While this notion of complexity can work at various levels, such as the number of significant digits for basic mathematical operations, the amount of memory and storage required, or the amount of time, we will typically focus on the time complexity.

For time complexity, the size $N$ is usually the dimensionality of the problem, although occasionally the key will be the number of non-zeros in the matrix or the width of bands. For our applications, time complexity is best thought of as the number of floating point operations (e.g., addition, multiplication) required.

### Notation

Complexity of algorithms is typically written in **Big O notation**, which provides bounds on the scaling of the computational complexity with respect to the size of the inputs.

Formally, if the number of operations required for a problem size $N$ is $f(N)$, we can write this as $f(N) = O(g(N))$ for some $g(N)$ - typically a polynomial.
The interpretation is that there exist some constants $M$ and $N_0$ such that

$$f(N) \leq Mg(N), \text{ for } N > N_0$$

For example, the complexity of finding an LU Decomposition of a dense matrix is $O(N^3)$, which should be read as there being a constant where eventually the number of floating point operations required to decompose a matrix of size $N \times N$ grows cubically.

Keep in mind that these are asymptotic results intended for understanding the scaling of the problem, and the constant can matter for a given fixed size.

For example, the number of operations required for an LU decomposition of a dense $N \times N$ matrix is $f(N) = \frac{2}{3}N^3$, ignoring the $N^2$ and lower terms. Other methods of solving a linear system may have different constants of proportionality, even if they have the same scaling, $O(N^3)$.

### 22.2.3 Rules of Computational Complexity

You will sometimes need to think through how combining algorithms changes complexity. For example, if you use

1. an $O(N^3)$ operation $P$ times, then it simply changes the constant. The complexity remains $O(N^3)$.

2. one $O(N^3)$ operation and one $O(N^2)$ operation, then you take the max. The complexity remains $O(N^3)$.

3. a repetition of an $O(N)$ operation that itself uses an $O(N)$ operation, you take the product. The complexity becomes $O(N^2)$.

With this, we have an important word of caution: Dense-matrix multiplication is an expensive operation for unstructured matrices. The naive version is $O(N^3)$ while the fastest-known algorithms (e.g., Coppersmith-Winograd) are roughly $O(N^{2.37})$. In practice, it is reasonable to crudely approximate with $O(N^3)$ when doing an analysis, in part since the higher constant factors of the better scaling algorithms dominate the better complexity until matrices become very large.

Of course, modern libraries use highly tuned and numerically stable algorithms to multiply matrices and exploit the computer architecture, memory cache, etc., but this simply lowers the constant of proportionality and they remain roughly approximated by $O(N^3)$.

A consequence is that, since many algorithms require matrix-matrix multiplication, it is often not possible to go below that order without further matrix structure.

That is, changing the constant of proportionality for a given size can help, but in order to achieve better scaling you need to identify matrix structure (e.g., tridiagonal, sparse) and ensure that your operations do not lose it.

### 22.2.4 Losing Structure

As a first example of a structured matrix, consider a sparse array.
In [3]: A = sprand(10, 10, 0.45)  # random sparse 10x10, 45 percent filled with non-zeros
   @show nnz(A)  # counts the number of non-zeros
   invA = sparse(inv(Array(A)))  # Julia won't invert sparse, so convert to dense with Array.
   @show nnz(invA);

   nnz(A) = 47
   nnz(invA) = 100

This increase from less than 50 to 100 percent dense demonstrates that significant sparsity can be lost when computing an inverse.

The results can be even more extreme. Consider a tridiagonal matrix of size $N \times N$ that might come out of a Markov chain or a discretization of a diffusion process.

In [4]: N = 5
   A = Tridiagonal([fill(0.1, N-2); 0.2], fill(0.8, N), [0.2; fill(0.1, N-2);])

Out[4]: 5x5 Tridiagonal{Float64,Array{Float64,1}}:
   0.8 0.2 0.0 0.0 0.0
   0.1 0.8 0.1 0.0 0.0
   0.0 0.1 0.8 0.1 0.0
   0.0 0.0 0.1 0.8 0.1
   0.0 0.0 0.0 0.2 0.8

The number of non-zeros here is approximately $3N$, linear, which scales well for huge matrices into the millions or billions

But consider the inverse

In [5]: inv(A)

Out[5]: 5x5 Array{Float64,2}:
   1.29099 -0.327957 0.0416667 -0.00537634 0.000672043
   -0.163978 1.31183 -0.166667 0.0215054 -0.00268817
   0.0208333 -0.166667 1.29167 -0.166667 0.0208333
   -0.00268817 0.0215054 -0.166667 1.31183 -0.163978
   0.000672043 -0.00537634 0.0416667 -0.327957 1.29099

Now, the matrix is fully dense and has $N^2$ non-zeros.

This also applies to the $A' A$ operation when forming the normal equations of linear least squares.

In [6]: A = sprand(20, 21, 0.3)
   @show nnz(A)/20^2
   @show nnz(A'*A)/21^2;

   nnz(A) / 20 ^ 2 = 0.2825
   nnz(A' * A) / 21 ^ 2 = 0.800453514739229
22.2. OVER VIEW

We see that a 30 percent dense matrix becomes almost full dense after the product is taken.

*Sparsity/Structure is not just for storage:* Matrix size can sometimes become important (e.g., a 1 million by 1 million tridiagonal matrix needs to store 3 million numbers (i.e., about 6MB of memory), where a dense one requires 1 trillion (i.e., about 1TB of memory)).

But, as we will see, the main purpose of considering sparsity and matrix structure is that it enables specialized algorithms, which typically have a lower computational order than unstructured dense, or even unstructured sparse, operations.

First, create a convenient function for benchmarking linear solvers

In [7]: using BenchmarkTools
   function benchmark_solve(A, b)
       println("A\b for typeof(A) = \\
        @btime $A \b $b
   end

Out[7]: benchmark_solve (generic function with 1 method)

Then, take away structure to see the impact on performance,

In [8]: N = 1000
   b = rand(N)
   A = Tridiagonal([fill(0.1, N-2); 0.2], fill(0.8, N), [0.2; fill(0.1, N-2);])
   A_sparse = sparse(A)  # sparse but losing tridiagonal structure
   A_dense = Array(A)    # dropping the sparsity structure, dense 1000x1000

   # benchmark solution to system A x = b
   benchmark_solve(A, b)
   benchmark_solve(A_sparse, b)
   benchmark_solve(A_dense, b);

   A\b for typeof(A) = Tridiagonal{Float64,Array{Float64,1}}
   32.810 μs (9 allocations: 47.75 KiB)
   A\b for typeof(A) = SparseMatrixCSC{Float64,Int64}
   778.201 μs (69 allocations: 1.06 MiB)
   A\b for typeof(A) = Array{Float64,2}
   32.689 ms (5 allocations: 7.65 MiB)

This example shows what is at stake: using a structured tridiagonal matrix may be 10-20 times faster than using a sparse matrix, which is 100 times faster than using a dense matrix.

In fact, the difference becomes more extreme as the matrices grow. Solving a tridiagonal system is $O(N)$, while that of a dense matrix without any structure is $O(N^3)$. The complexity of a sparse solution is more complicated, and scales in part by the $\text{nnz}(N)$, i.e., the number of nonzeros.

22.2.5 Matrix Multiplication

While we write matrix multiplications in our algebra with abundance, in practice the computational operation scales very poorly without any matrix structure.
Matrix multiplication is so important to modern computers that the constant of scaling is small using proper packages, but the order is still roughly $O(N^3)$ in practice (although smaller in theory, as discussed above).

Sparse matrix multiplication, on the other hand, is $O(N M_A M_B)$ where $M_A$ is the number of nonzeros per row of $A$ and $M_B$ is the number of non-zeros per column of $B$.

By the rules of computational order, that means any algorithm requiring a matrix multiplication of dense matrices requires at least $O(N^3)$ operation.

The other important question is what is the structure of the resulting matrix. For example, multiplying an upper triangular matrix by a lower triangular matrix

```
N = 5
U = UpperTriangular(rand(N,N))
```

```
Out[9]: 5×5 UpperTriangular{Float64,Array{Float64,2}}:
    0.299976  0.176934  0.0608682  0.20465  0.409653
    0.523923  0.127154  0.512531  0.235328
    0.600588  0.682868  0.330638
    0.345419  0.0312986
    0.471043
```

```
L = U'
```

```
Out[10]: 5×5 Adjoint{Float64,UpperTriangular{Float64,Array{Float64,2}}}:
    0.299976  0.0  0.0  0.0  0.0
    0.176934  0.523923  0.0  0.0  0.0
    0.0608682  0.127154  0.600588  0.0  0.0
    0.20465  0.512531  0.682868  0.345419  0.0
    0.409653  0.235328  0.330638  0.0312986  0.471043
```

But the product is fully dense (e.g., think of a Cholesky multiplied by itself to produce a covariance matrix)

```
L = U
```

```
Out[11]: 5×5 Array{Float64,2}:
    0.0899855  0.0530758  0.018259  0.0613901  0.122886
    0.0530758  0.305801  0.0773883  0.304736  0.195775
    0.018259  0.0773883  0.380579  0.487749  0.253435
    0.0613901  0.304736  0.487749  0.890193  0.441042
    0.122886  0.195775  0.253435  0.441042  0.555378
```

On the other hand, a tridiagonal matrix times a diagonal matrix is still tridiagonal - and can use specialized $O(N)$ algorithms.

```
A = Tridiagonal([fill(0.1, N-2); 0.2], fill(0.8, N), [0.2; fill(0.1, N-2)])
D = Diagonal(rand(N))
D * A
```

22.3. FACTORIZATIONS

When you tell a numerical analyst you are solving a linear system using direct methods, their first question is “which factorization?”.

Just as you can factor a number (e.g., $6 = 3 \times 2$) you can factor a matrix as the product of other, more convenient matrices (e.g., $A = LU$ or $A = QR$, where $L, U, Q$, and $R$ have properties such as being triangular, orthogonal, etc.).

22.3.1 Inverting Matrices

On paper, since the Invertible Matrix Theorem tells us that a unique solution is equivalent to $A$ being invertible, we often write the solution to $Ax = b$ as

$$x = A^{-1}b$$

What if we do not (directly) use a factorization?

Take a simple linear system of a dense matrix,

In [13]: N = 4
A = rand(N,N)
b = rand(N)

Out[13]: 4-element Array{Float64,1}:
0.5682240701809245
0.40245385575255055
0.1825995192132288
0.06160128039631019

On paper, we try to solve the system $Ax = b$ by inverting the matrix,

In [14]: x = inv(A) * b

Out[14]: 4-element Array{Float64,1}:
-0.0339069840407679
0.7988200873225003
0.9963711951331815
-0.9276352098500461

As we will see throughout, inverting matrices should be used for theory, not for code. The classic advice that you should never invert a matrix may be slightly exaggerated, but is generally good advice.
Solving a system by inverting a matrix is always a little slower, is potentially less accurate, and will sometimes lose crucial sparsity compared to using factorizations. Moreover, the methods used by libraries to invert matrices are frequently the same factorizations used for computing a system of equations.

Even if you need to solve a system with the same matrix multiple times, you are better off factoring the matrix and using the solver rather than calculating an inverse.

```
In [15]: N = 100
A = rand(N, N)
M = 30
B = rand(N, M)

function solve_inverting(A, B)
    A_inv = inv(A)
    X = similar(B)
    for i in 1:size(B, 2)
        X[:,i] = A_inv * B[:,i]
    end
    return X
end

function solve_factoring(A, B)
    X = similar(B)
    A = factorize(A)
    for i in 1:size(B, 2)
        X[:,i] = A \ B[:,i]
    end
    return X
end

@btime solve_inverting($A, $B)
@btime solve_factoring($A, $B)

# even better, use the built-in feature for multiple RHS
@btime $A \ $B;
```

499.932 μs (68 allocations: 205.28 KiB)
421.385 μs (96 allocations: 155.59 KiB)
245.005 μs (6 allocations: 102.63 KiB)

### 22.3.2 Triangular Matrices and Back/Forward Substitution

Some matrices are already in a convenient form and require no further factoring. For example, consider solving a system with an `UpperTriangular` matrix,

```
In [16]: b = [1.0, 2.0, 3.0]
U = UpperTriangular([[1.0 2.0 3.0; 0.0 5.0 6.0; 0.0 0.0 9.0]])

Out[16]: 3×3 UpperTriangular{Float64,Array{Float64,2}}:
    1.0 2.0 3.0
    ⤵ 5.0 6.0
    ⤵ ⤵ 9.0
```
This system is especially easy to solve using back substitution. In particular, \( x_3 = b_3/U_{33}, x_2 = (b_2 - x_3 U_{23})/U_{22}, \) etc.

In [17]: U \ b

Out[17]: 3-element Array{Float64,1}:
0.0
0.0
0.3333333333333333

A LowerTriangular matrix has similar properties and can be solved with forward substitution.

The computational order of back substitution and forward substitution is \( O(N^2) \) for dense matrices. Those fast algorithms are a key reason that factorizations target triangular structures.

### 22.3.3 LU Decomposition

The \( LU \) decomposition finds a lower triangular matrix \( L \) and an upper triangular matrix \( U \) such that \( LU = A \).

For a general dense matrix without any other structure (i.e., not known to be symmetric, tridiagonal, etc.) this is the standard approach to solve a system and exploit the speed of back and forward substitution using the factorization.

The computational order of LU decomposition itself for a dense matrix is \( O(N^3) \) - the same as Gaussian elimination - but it tends to have a better constant term than others (e.g., half the number of operations of the QR decomposition). For structured or sparse matrices, that order drops.

We can see which algorithm Julia will use for the \( \backslash \) operator by looking at the \texttt{factorize} function for a given matrix.

In [18]: N = 4
   A = rand(N,N)
   b = rand(N)

   Af = factorize(A) # chooses the right factorization, LU here

Out[18]: LU{Float64,Array{Float64,2}}

L factor:
4×4 Array{Float64,2}:
1.0  0.0  0.0  0.0
0.563082 1.0  0.0  0.0
0.730109 0.912509 1.0  0.0
0.114765 0.227879 0.115228 1.0
U factor:
4×4 Array{Float64,2}:
0.79794  0.28972  0.765939  0.496278
0.0  0.82524  0.23962 -0.130989
0.0  0.0  -0.447888  0.374303
0.0  0.0  0.0  0.725264
In this case, it provides an $L$ and $U$ factorization (with pivoting).

With the factorization complete, we can solve different $b$ right hand sides.

In [19]: $\text{Af} \backslash \text{b}$

Out[19]: 4-element Array{Float64,1}:
-0.49842605495731557
-0.11835721499695576
1.5055538550184817
0.07694455957797537

In [20]: $b_2 = \text{rand}(N)$
$\text{Af} \backslash b_2$

Out[20]: 4-element Array{Float64,1}:
-0.6456780666059364
-0.2601515737654759
1.11689566296631
0.5405293106660054

In practice, the decomposition also includes a $P$ which is a permutation matrix such that $PA = LU$.

In [21]: $\text{Af.P} * A \approx \text{Af.L} * \text{Af.U}$

Out[21]: True

We can also directly calculate an LU decomposition with $\text{lu}$ but without the pivoting,

In [22]: $L, U = \text{lu}(A, \text{Val(false)})$ # the $\text{Val(false)}$ provides a solution without permutation matrices

Out[22]: LU{Float64,Array{Float64,2}}
  L factor:
  4×4 Array{Float64,2}:
  1.0  0.0  0.0  0.0
  0.730109 1.0  0.0  0.0
  0.563082 1.09588 1.0  0.0
  0.114765 0.249728 0.122733 1.0
  U factor:
  4×4 Array{Float64,2}:
  0.79794  0.28972  0.765939  0.496278
  0.0  0.753039 -0.229233  0.254774
  0.0  0.0  0.490832 -0.410191
  0.0  0.0  0.0  0.725264

And we can verify the decomposition

In [23]: $A \approx L * U$
To see roughly how the solver works, note that we can write the problem \( Ax = b \) as \( L U x = b \). Let \( U x = y \), which breaks the problem into two sub-problems.

\[
Ly = b \\
Ux = y
\]

As we saw above, this is the solution to two triangular systems, which can be efficiently done with back or forward substitution in \( O(N^2) \) operations.

To demonstrate this, first using

\[
\text{In [24]: } y = L \backslash b
\]

\[
\text{Out[24]: 4-element Array{Float64,1}:} \\
0.759344042755733 \\
-0.4146467815590597 \\
0.707411438334498 \\
0.05580508465599857
\]

\[
\text{In [25]: } x = U \backslash y \\
x \approx A \backslash b \quad \# \text{Check identical}
\]

\[
\text{Out[25]: true}
\]

The LU decomposition also has specialized algorithms for structured matrices, such as a Tridiagonal

\[
\text{In [26]: } \begin{align*}
N &= 1000 \\
b &= \text{rand}(N) \\
A &= \text{Tridiagonal}([\text{fill}(0.1, N-2); 0.2], \text{fill}(0.8, N), [0.2; \text{fill}(0.1, N-2)]; \text{fill}(0.1, N-2)); \\
\end{align*}
\]

\[
\text{factorize}(A) \mid > \text{typeof}
\]

\[
\text{Out[26]: } \text{LU}{\text{Float64,Tridiagonal}}{\text{Float64,Array}}{\text{Float64,1}}}
\]

This factorization is the key to the performance of the \( A \backslash b \) in this case. For Tridiagonal matrices, the LU decomposition is \( O(N^2) \).

Finally, just as a dense matrix without any structure uses an LU decomposition to solve a system, so will the sparse solvers

\[
\text{In [27]: } A\_\text{sparse} = \text{sparse}(A) \\
\text{factorize}(A\_\text{sparse}) \mid > \text{typeof} \quad \# \text{dropping the tridiagonal structure to} \\
\text{just become}
\]

\[
\text{sparse}
\]

\[
\text{Out[27]: } \text{SuiteSparse.UMFPACK.UmfpackLU}{\text{Float64,Int64}}}
\]

\[
\text{In [28]: } \text{benchmark\_solve}(A, b) \\
\text{benchmark\_solve}(A\_\text{sparse}, b);
\]
With sparsity, the computational order is related to the number of non-zeros rather than the size of the matrix itself.

### 22.3.4 Cholesky Decomposition

For real, symmetric, positive semi-definite matrices, a Cholesky decomposition is a specialized example of an LU decomposition where $L = U'$.

The Cholesky is directly useful on its own (e.g., Classical Control with Linear Algebra), but it is also an efficient factorization to use in solving symmetric positive semi-definite systems.

As always, symmetry allows specialized algorithms.

```
In [29]: N = 500
   B = rand(N,N)
   A_dense = B' * B  # an easy way to generate a symmetric positive semi-
   definite matrix
   A = Symmetric(A_dense)  # flags the matrix as symmetric

   factorize(A) |> typeof

Out[29]: BunchKaufman{Float64,Array{Float64,2}}
```

Here, the $A$ decomposition is Bunch-Kaufman rather than Cholesky, because Julia doesn’t know that the matrix is positive semi-definite. We can manually factorize with a Cholesky,

```
In [30]: cholesky(A) |> typeof

Out[30]: Cholesky{Float64,Array{Float64,2}}
```

Benchmarking,

```
In [31]: b = rand(N)
   cholesky(A) \ b  # use the factorization to solve

   benchmark_solve(A, b)
   benchmark_solve(A_dense, b)
   @btime cholesky($A, check=false) \ $b;

   A\b for typeof(A) = Symmetric{Float64,Array{Float64,2}}
   5.748 ms (8 allocations: 2.16 MiB)
   A\b for typeof(A) = Array{Float64,2}
   8.141 ms (5 allocations: 1.92 MiB)
   3.865 ms (7 allocations: 1.91 MiB)
```
### 22.3.5 QR Decomposition

Previously, we learned about applications of the QR decomposition to solving the linear least squares.

While in principle the solution to the least-squares problem

$$\min_x \|Ax - b\|^2$$

is $x = (A'A)^{-1}A'b$, in practice note that $A'A$ becomes dense and calculating the inverse is rarely a good idea.

The QR decomposition is a decomposition $A = QR$ where $Q$ is an orthogonal matrix (i.e., $Q'Q = QQ' = I$) and $R$ is an upper triangular matrix.

Given the previous derivation, we showed that we can write the least-squares problem as the solution to

$$Rx = Q'b$$

where, as discussed above, the upper-triangular structure of $R$ can be solved easily with back substitution.

The `\` operator solves the linear least-squares problem whenever the given $A$ is rectangular.

In [32]:
N = 10
M = 3
x_true = rand(3)
A = rand(N,M) .+ randn(N)
b = rand(N)
x = A \ b

Out[32]:
3-element Array{Float64,1}:
  0.4011747124872585
  0.07361080010718478
  -0.2347806801272457

To manually use the QR decomposition in solving linear least squares:

In [33]:
Af = qr(A)
Q = Af.Q
R = [Af.R; zeros(N - M, M)] # Stack with zeros
@show Q * R ≈ A
x = R \ Q'*b  # simplified QR solution for least squares

Q * R ≈ A = true

Out[33]:
3-element Array{Float64,1}:
  0.4011747124872585
  0.07361080010718478
  -0.2347806801272457
This stacks the $R$ with zeros, but the more specialized algorithm would not multiply directly in that way.

In some cases, if an LU is not available for a particular matrix structure, the QR factorization can also be used to solve systems of equations (i.e., not just LLS). This tends to be about 2 times slower than the LU but is of the same computational order.

Deriving the approach, where we can now use the inverse since the system is square and we assumed $A$ was non-singular,

$$Ax = b$$
$$QRx = b$$
$$Q^{-1}QRx = Q^{-1}b$$
$$Rx = Q'b$$

where the last step uses the fact that $Q^{-1} = Q'$ for an orthogonal matrix.

Given the decomposition, the solution for dense matrices is of computational order $O(N^2)$. To see this, look at the order of each operation.

- Since $R$ is an upper-triangular matrix, it can be solved quickly through back substitution with computational order $O(N^2)$
- A transpose operation is of order $O(N^2)$
- A matrix-vector product is also $O(N^2)$

In all cases, the order would drop depending on the sparsity pattern of the matrix (and corresponding decomposition). A key benefit of a QR decomposition is that it tends to maintain sparsity.

Without implementing the full process, you can form a QR factorization with `qr` and then use it to solve a system

```
In [34]: N = 5
   : A = rand(N,N)
   : b = rand(N)
   : @show A \ b
   : @show qr(A) \ b;

A \ b = [-1.478040941944558, 2.09875752634393, -0.6857071090150306, 
   : -0.16849538664184543, 2.012803045177841]
   : qr(A) \ b = [-1.4780409419445582, 2.09875752634393, -0.685707109015032, 
   : -0.16849538664184413, 2.0128030451778414]
```

### 22.3.6 Spectral Decomposition

A spectral decomposition, also known as an eigendecomposition, finds all of the eigenvectors and eigenvalues to decompose a square matrix $A$ such that

$$A = QΛQ^{-1}$$

where $Q$ is a matrix made of the eigenvectors of $A$ as columns, and $Λ$ is a diagonal matrix of the eigenvalues. Only square, diagonalizable matrices have an eigendecomposition (where a matrix is not diagonalizable if it does not have a full set of linearly independent eigenvectors).
In Julia, whenever you ask for a full set of eigenvectors and eigenvalues, it decomposes using an algorithm appropriate for the matrix type. For example, symmetric, Hermitian, and tridiagonal matrices have specialized algorithms.

To see this,

```julia
In [35]: A = Symmetric(rand(5, 5)) # symmetric matrices have real eigenvectors/eigenvalues
A_eig = eigen(A)
Λ = Diagonal(A_eig.values)
Q = A_eig.vectors
norm(Q * Λ * inv(Q) - A)
Out[35]: 2.803627108839096e-15
```

Keep in mind that a real matrix may have complex eigenvalues and eigenvectors, so if you attempt to check $Q \times Λ \times inv(Q) - A$ - even for a positive-definite matrix - it may not be a real number due to numerical inaccuracy.

### 22.4 Continuous-Time Markov Chains (CTMCs)

In the previous lecture on discrete-time Markov chains, we saw that the transition probability between state $x$ and state $y$ was summarized by the matrix $P(x, y) := P\{X_{t+1} = y \mid X_t = x\}$.

As a brief introduction to continuous time processes, consider the same state space as in the discrete case: $S$ is a finite set with $n$ elements $\{x_1, \ldots, x_n\}$.

A Markov chain $\{X_t\}$ on $S$ is a sequence of random variables on $S$ that have the Markov property.

In continuous time, the Markov Property is more complicated, but intuitively is the same as the discrete-time case.

That is, knowing the current state is enough to know probabilities for future states. Or, for realizations $x(\tau) \in S, \tau \leq t$,

$$P\{X(t + s) = y \mid X(t) = x, X(\tau) = x(\tau) \text{ for } 0 \leq \tau \leq t\} = P\{X(t + s) = y \mid X(t) = x\}$$

Heuristically, consider a time period $t$ and a small step forward, $\Delta$. Then the probability to transition from state $i$ to state $j$ is

$$P\{X(t + \Delta) = j \mid X(t)\} = \begin{cases} q_{ij}\Delta + o(\Delta) & i \neq j \\ 1 + q_{ii}\Delta + o(\Delta) & i = j \end{cases}$$

where the $q_{ij}$ are “intensity” parameters governing the transition rate, and $o(\Delta)$ is little-o notation. That is, $\lim_{\Delta \to 0} o(\Delta)/\Delta = 0$.

Just as in the discrete case, we can summarize these parameters by an $N \times N$ matrix, $Q \in R^{N \times N}$.

Recall that in the discrete case every element is weakly positive and every row must sum to one. With continuous time, however, the rows of $Q$ sum to zero, where the diagonal contains the negative value of jumping out of the current state. That is,
• $q_{ij} \geq 0$ for $i \neq j$
• $q_{ii} \leq 0$
• $\sum_j q_{ij} = 0$

The $Q$ matrix is called the intensity matrix, or the infinitesimal generator of the Markov chain. For example,

$$Q = \begin{bmatrix}
-0.1 & 0.1 & 0 & 0 & 0 & 0 \\
0.1 & -0.2 & 0.1 & 0 & 0 & 0 \\
0 & 0.1 & -0.2 & 0.1 & 0 & 0 \\
0 & 0 & 0.1 & -0.2 & 0.1 & 0 \\
0 & 0 & 0 & 0.1 & -0.2 & 0.1 \\
0 & 0 & 0 & 0 & 0.1 & -0.1 \\
\end{bmatrix}$$

In the above example, transitions occur only between adjacent states with the same intensity (except for a “bouncing back” of the bottom and top states).

Implementing the $Q$ using its tridiagonal structure

In [36]: using LinearAlgebra

\[
\alpha = 0.1
\]
\[
N = 6
\]
\[
Q = \text{Tridiagonal}(\text{fill}(\alpha, N-1), [-\alpha; \text{fill}(-2\alpha, N-2); -\alpha], \text{fill}(\alpha, N-1))
\]

Out[36]: $6 \times 6$ Tridiagonal{$\text{Float64}, \text{Array{Float64,1}}$}:

$$
\begin{array}{ccccccc}
-0.1 & 0.1 & \cdot & \cdot & \cdot & \cdot & \\
0.1 & -0.2 & 0.1 & \cdot & \cdot & \cdot & \\
\cdot & 0.1 & -0.2 & 0.1 & \cdot & \cdot & \\
\cdot & \cdot & 0.1 & -0.2 & 0.1 & \cdot & \\
\cdot & \cdot & \cdot & 0.1 & -0.2 & 0.1 & \\
\cdot & \cdot & \cdot & \cdot & 0.1 & -0.1 & \\
\end{array}
$$

Here we can use \text{Tridiagonal} to exploit the structure of the problem.

Consider a simple payoff vector $r$ associated with each state, and a discount rate $\rho$. Then we can solve for the expected present discounted value in a way similar to the discrete-time case.

$$\rho v = r + Qv$$

or rearranging slightly, solving the linear system

$$(\rho I - Q)v = r$$

For our example, exploiting the tridiagonal structure,

In [37]: $r = \text{range}(0.0, 10.0, \text{length}=N)$

\[
\rho = 0.05
\]

\[
A = \rho * I - Q
\]

Out[37]: $6 \times 6$ Tridiagonal{$\text{Float64}, \text{Array{Float64,1}}$}:

$$
\begin{array}{ccccccc}
0.15 & -0.1 & \cdot & \cdot & \cdot & \cdot & \\
-0.1 & 0.25 & -0.1 & \cdot & \cdot & \cdot & \\
\end{array}
$$
Note that this \( A \) matrix is maintaining the tridiagonal structure of the problem, which leads to an efficient solution to the linear problem.

In [38]: \( v = A \backslash r \)

Out[38]: 6-element Array{Float64,1}:
38.15384615384615
57.23076923076923
84.92307692307693
115.07692307692311
142.76923076923077
161.84615384615384

The \( Q \) is also used to calculate the evolution of the Markov chain, in direct analogy to the \( \psi_{t+k} = \psi_t P^k \) evolution with the transition matrix \( P \) of the discrete case.

In the continuous case, this becomes the system of linear differential equations

\[
\dot{\psi}(t) = Q(t)^T \psi(t)
\]

given the initial condition \( \psi(0) \) and where the \( Q(t) \) intensity matrix is allowed to vary with time. In the simplest case of a constant \( Q \) matrix, this is a simple constant-coefficient system of linear ODEs with coefficients \( Q^T \).

If a stationary equilibrium exists, note that \( \dot{\psi}(t) = 0 \), and the stationary solution \( \psi^* \) needs to satisfy

\[
0 = Q^T \psi^*
\]

Notice that this is of the form \( 0 \psi^* = Q^T \psi^* \) and hence is equivalent to finding the eigenvector associated with the \( \lambda = 0 \) eigenvalue of \( Q^T \).

With our example, we can calculate all of the eigenvalues and eigenvectors

In [39]: \( \lambda, \text{vecs} = \text{eigen}(\text{Array}(Q')) \)

Out[39]:
Eigen{Float64,Float64,Array{Float64,2},Array{Float64,1}}
values:
6-element Array{Float64,1}:
-0.3732058087568874
-0.29999999999999993
-0.19999999999999998
-0.09999999999999995
-0.02679491924311274
0.0
vectors:
6×6 Array{Float64,2}:
-0.149429 -0.288675 0.408248 0.5 -0.557678 0.408248
Indeed, there is a $\lambda = 0$ eigenvalue, which is associated with the last column in the eigenvector. To turn that into a probability, we need to normalize it.

```
In [40]: vecs[:,N] ./ sum(vecs[:,N])
```

```
Out[40]: 6-element Array{Float64,1}:
 0.16666666666666657
 0.16666666666666657
 0.16666666666666667
 0.16666666666666682
 0.16666666666666685
 0.16666666666666663
```

### 22.4.1 Multiple Dimensions

A frequent case in discretized models is dealing with Markov chains with multiple “spatial” dimensions (e.g., wealth and income).

After discretizing a process to create a Markov chain, you can always take the Cartesian product of the set of states in order to enumerate it as a single state variable.

To see this, consider states $i$ and $j$ governed by infinitesimal generators $Q$ and $A$.

```
In [41]: function markov_chain_product(Q, A)
    M = size(Q, 1)
    N = size(A, 1)
    Q = sparse(Q)
    Qs = blockdiag(fill(Q, N)... # create diagonal blocks of every
    As = kron(A, sparse(I(M)))
    return As + Qs
end

α = 0.1
N = 4
Q = Tridiagonal(fill(α, N-1), [-α; fill(-2α, N-2); -α], fill(α, N-1))
A = sparse([-0.1 0.1
    0.2 -0.2])
M = size(A,1)
L = markov_chain_product(Q, A)
L |> Matrix # display as a dense matrix
```

```
Out[41]: 8×8 Array{Float64,2}:
 -0.2  0.1  0.0  0.0  0.1  0.0  0.0  0.0
  0.1 -0.3  0.1  0.0  0.0  0.1  0.0  0.0
  0.0  0.1 -0.3  0.1  0.0  0.0  0.1  0.0
  0.0  0.0  0.1 -0.2  0.0  0.0  0.0  0.1
  0.2  0.0  0.0  0.0 -0.3  0.1  0.0  0.0
  0.0  0.2  0.0  0.0  0.1 -0.4  0.1  0.0
```
22.4. CONTINUOUS-TIME MARKOV CHAINS (CTMCS)

\[
\begin{pmatrix}
0.0 & 0.0 & 0.2 & 0.0 & 0.0 & 0.1 & -0.4 & 0.1 \\
0.0 & 0.0 & 0.0 & 0.2 & 0.0 & 0.0 & 0.1 & -0.3
\end{pmatrix}
\]

This provides the combined Markov chain for the \((i, j)\) process. To see the sparsity pattern,

\[
\text{In [42]: using Plots} \\
\text{gr(fmt = :png);} \\
\text{spy(L, markersize = 10)}
\]

\[
\text{Out[42]:}
\]

\[
\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}
\]

To calculate a simple dynamic valuation, consider whether the payoff of being in state \((i, j)\) is \(r_{ij} = i + 2j\)

\[
\text{In [43]: } r = [i + 2.0j \text{ for } i \text{ in } 1:N, j \text{ in } 1:M] \\
r = \text{vec}(r) \quad \# \text{ vectorize it since stacked in same order}
\]

\[
\text{Out[43]: 8-element Array{Float64,1}:} \\
3.0 \\
4.0 \\
5.0 \\
6.0 \\
5.0 \\
6.0 \\
7.0 \\
8.0
\]

Solving the equation \(\rho v = r + Lv\)
\[\rho = 0.05\]
\[v = (\rho \ast I - L) \ \text{\textbackslash} \ r\]
\[\text{reshape}(v, N, M)\]

```
In [44]:
\[\text{Out[44]: 4\times2 Array{Float64,2}}:\]
0.878992 0.936134
0.961345 1.01849
1.06723 1.12437
1.14958 1.120672
```

The \texttt{reshape} helps to rearrange it back to being two-dimensional.

To find the stationary distribution, we calculate the eigenvalue and choose the eigenvector associated with \(\lambda = 0\). In this case, we can verify that it is the last one.

```
In [45]:
\[\text{Out[45]: 8-element Array{Float64,1}}:\]
0.16666666666666677
0.1666666666666665
0.16666666666666682
0.16666666666666666
0.08333333333333325
0.08333333333333345
0.0833333333333333
0.0833333333333334
```

22.4.2 Irreducibility

As with the discrete-time Markov chains, a key question is whether CTMCs are reducible, i.e., whether states communicate. The problem is isomorphic to determining whether the directed graph of the Markov chain is strongly connected.

```
In [47]:
\[\text{Out[47]: 4\times2 Array{Float64,2}}:\]
0.166667 0.0833333
0.166667 0.0833333
0.166667 0.0833333
0.166667 0.0833333
```

\[\text{using LightGraphs}\]
\[\alpha = 0.1\]
\[N = 6\]
\[Q = \text{Tridiagonal}(\text{fill}(\alpha, N-1), [-\alpha; \text{fill}(-2\alpha, N-2); -\alpha], \text{fill}(\alpha, N-1))\]
22.5. **BANDED MATRICES**

A tridiagonal matrix has 3 non-zero diagonals: the main diagonal, the first sub-diagonal (i.e., below the main diagonal), and also the first super-diagonal (i.e., above the main diagonal).

This is a special case of a more general type called a banded matrix, where the number of sub- and super-diagonals can be greater than 1. The total width of main-, sub-, and super-diagonals is called the bandwidth. For example, a tridiagonal matrix has a bandwidth of 3.

An $N \times N$ banded matrix with bandwidth $P$ has about $NP$ nonzeros in its sparsity pattern.

These can be created directly as a dense matrix with `diagm`. For example, with a bandwidth of three and a zero diagonal,

```
In [50]: diagm(1 => [1,2,3], -1 => [4,5,6])
```

```
Out[50]: 4×4 Array{Int64,2}:
0 1 0 0
4 0 2 0
0 5 0 3
0 0 6 0
```
Or as a sparse matrix,

In [51]: `spdiagm(1 => [1,2,3], -1 => [4,5,6])`

Out[51]: 4×4 SparseMatrixCSC{Int64,Int64} with 6 stored entries:

```
[2, 1] = 4
[1, 2] = 1
[3, 2] = 5
[2, 3] = 2
[4, 3] = 6
[3, 4] = 3
```

Or directly using `BandedMatrices.jl`

In [52]: ```
using BandedMatrices
BandedMatrix(1 => [1,2,3], -1 => [4,5,6])
```

Out[52]: 4×4 BandedMatrix{Int64,Array{Int64,2},Base.OneTo{Int64}}:

```
0 1 0 0
4 0 2 0
5 0 3 0
6 0 0 0
```

There is also a convenience function for generating random banded matrices

In [53]: ```
A = brand(7, 7, 3, 1) # 7×7 matrix, 3 subdiagonals, 1 superdiagonal
```

Out[53]: 7×7 BandedMatrix{Float64,Array{Float64,2},Base.OneTo{Int64}}:

```
 0.386608 0.346262 0.479616 0.857743 0.779364 0.793664 0.386608
 0.346262 0.479616 0.750276 0.913585 0.293782 0.034123 0.346262
 0.479616 0.750276 0.50362 0.913585 0.293782 0.47962 0.750276
 0.857743 0.913585 0.087322 0.50362 0.293782 0.857743 0.913585
 0.779364 0.293782 0.269804 0.087322 0.034123 0.346262 0.479616
 0.793664 0.034123 0.711412 0.438157 0.031229 0.346262 0.479616
 0.386608 0.47962 0.750276 0.857743 0.779364 0.793664 0.386608
```

And, of course, specialized algorithms will be used to exploit the structure when solving linear systems. In particular, the complexity is related to the $O(NP_L P_U)$ for upper and lower bandwidths $P$

In [54]: ```
@show factorize(Symmetric(A)) |> typeof
A \ rand(7)
```

```
factorize(Symmetric(A)) |> typeof =
LDLt{Float64,Symmetric{Float64,BandedMatrix{Float64,Array{Float64,2},Base.OneTo{Int64}}}}
```

Out[54]: 7-element Array{Float64,1}:

```
-0.6345917189136551
1.2689275835805582
0.5499404721793494
0.24947160343942412
-0.45227412611006496
0.4973200025591808
1.3752489574369149
```
The factorization algorithm uses a specialized LU decomposition for banded matrices.

22.6 Implementation Details and Performance

Recall the famous quote from Knuth: “97% of the time, premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.” The most common example of premature optimization is trying to use your own mental model of a compiler while writing your code, worried about the efficiency of code, and (usually incorrectly) second-guessing the compiler.

Concretely, the lessons in this section are:

1. Don’t worry about optimizing your code unless you need to. Code clarity is your first-order concern.
2. If you use other people’s packages, they can worry about performance and you don’t need to.
3. If you absolutely need that “critical 3%,” your intuition about performance is usually wrong on modern CPUs and GPUs, so let the compiler do its job.
4. Benchmarking (e.g., @btime) and profiling are the tools to figure out performance bottlenecks. If 99% of computing time is spent in one small function, then there is no point in optimizing anything else.
5. If you benchmark to show that a particular part of the code is an issue, and you can’t find another library that does a better job, then you can worry about performance.

You will rarely get to step 3, let alone step 5.

However, there is also a corollary: “don’t pessimize prematurely.” That is, don’t make choices that lead to poor performance without any tradeoff in improved code clarity. For example, writing your own algorithms when a high-performance algorithm exists in a package or Julia itself, or lazily making a matrix dense and carelessly dropping its structure.

22.6.1 Implementation Difficulty

Numerical analysts sometimes refer to the lowest level of code for basic operations (e.g., a dot product, matrix-matrix product, convolutions) as kernels.

That sort of code is difficult to write, and performance depends on the characteristics of the underlying hardware, such as the instruction set available on the particular CPU, the size of the CPU cache, and the layout of arrays in memory.

Typically, these operations are written in a BLAS library, organized into different levels. The levels roughly correspond to the computational order of the operations: BLAS Level 1 are $O(N)$ operations such as linear products, Level 2 are $O(N^2)$ operations such as matrix-vector products, and Level 3 are roughly $O(N^3)$, such as general matrix-matrix products.

An example of a BLAS library is OpenBLAS, which is used by default in Julia, or the Intel MKL, which is used in Matlab (and in Julia if the MKL.jl package is installed).
On top of BLAS are LAPACK operations, which are higher-level kernels, such as matrix factorizations and eigenvalue algorithms, and are often in the same libraries (e.g., MKL has both BLAS and LAPACK functionality).

The details of these packages are not especially relevant, but if you are talking about performance, people will inevitably start discussing these different packages and kernels. There are a few important things to keep in mind:

1. Leave writing kernels to the experts. Even simple-sounding algorithms can be very complicated to implement with high performance.

2. Your intuition about performance of code is probably going to be wrong. If you use high quality libraries rather than writing your own kernels, you don’t need to use your intuition.

3. Don’t get distracted by the jargon or acronyms above if you are reading about performance.

22.6.2 Row- and Column-Major Ordering

There is a practical performance issue which may influence your code. Since memory in a CPU is linear, dense matrices need to be stored by either stacking columns (called column-major order) or rows.

The reason this matters is that compilers can generate better performance if they work in contiguous chunks of memory, and this becomes especially important with large matrices due to the interaction with the CPU cache. Choosing the wrong order when there is no benefit in code clarity is an example of premature pessimization. The performance difference can be orders of magnitude in some cases, and nothing in others.

One option is to use the functions that let the compiler choose the most efficient way to traverse memory. If you need to choose the looping order yourself, then you might want to experiment with going through columns first and going through rows first. Other times, let Julia decide, i.e., enumerate and eachindex will choose the right approach.

Julia, Fortran, and Matlab all use column-major order, while C/C++ and Python use row-major order. This means that if you find an algorithm written for C/C++/Python, you will sometimes need to make small changes if performance is an issue.

22.6.3 Digression on Allocations and In-place Operations

While we have usually not considered optimizing code for performance (and have focused on the choice of algorithms instead), when matrices and vectors become large we need to be more careful.

The most important thing to avoid are excess allocations, which usually occur due to the use of temporary vectors and matrices when they are not necessary. Sometimes those extra temporary values can cause enormous degradations in performance.

However, caution is suggested since excess allocations are never relevant for scalar values, and allocations frequently create faster code for smaller matrices/vectors since it can lead to better cache locality.

To see this, a convenient tool is the benchmarking
In [55]: using BenchmarkTools
A = rand(10,10)
B = rand(10,10)
C = similar(A)
definition f!(C, A, B)
    D = A*B
    C .= D + 1
end
@btime f!(C, A, B)

535.183 ns (1 allocation: 896 bytes)

Out[55]: 10×10 Array{Float64,2}:
```
3.26172 4.24569 3.37182 4.10324 ... 4.03344 4.39198 2.88797 2.63934
4.19687 4.58126 3.88015 5.0409   ... 4.0105 3.56832 2.35475 3.32362
2.17535 2.58069 3.08736 3.04461   2.71563 3.03535 2.62734 2.37854
3.16928 4.20751 3.08482 3.89843   3.81516 4.14681 2.64178 2.94961
3.01031 3.08903 2.83417 3.80852   3.22832 3.29357 2.57282 2.60746
2.7023  3.10147 2.95828 3.63363   3.64397 3.40609 2.44341 3.83272
3.02687 3.13864 2.78748 3.90634   3.18422 2.90128 1.99457 2.80653
3.80929 3.83931 3.88255 4.8596    4.16155 3.73634 2.65279 3.07034
```

The ! on the f! is an informal way to say that the function is mutating, and the first argument (C here) is by convention the modified variable.

In the f! function, notice that the D is a temporary variable which is created, and then modified afterwards. But notice that since C is modified directly, there is no need to create the temporary D matrix.

This is an example of where an in-place version of the matrix multiplication can help avoid the allocation.

In [56]: function f2!(C, A, B)
    mul!(C, A, B) # in-place multiplication
    C .+= 1
end
A = rand(10,10)
B = rand(10,10)
C = similar(A)
@btime f!(C, A, B)
@btime f2!(C, A, B)

539.402 ns (1 allocation: 896 bytes)
498.782 ns (0 allocations: 0 bytes)

Out[56]: 10×10 Array{Float64,2}:
```
2.42733 3.74571 2.5811 2.745   ... 2.45258 3.17339 2.792  3.46213
3.52188 4.16932 3.17155 3.98401 2.1202 2.85629 3.35848 3.88871
2.47181 4.33343 2.46863 2.68593 2.38238 3.6709  3.2434  4.17783
```
3.29888  5.38813  3.4278  3.8622  2.87482  4.07336  3.89498  5.41919
2.96602  3.60521  3.2117  3.0913  2.68528  2.99728  3.34362  3.47657
4.73208  5.38525  4.42378  5.18235  2.91664  4.70184  5.28638  5.4401

Note that in the output of the benchmarking, the `f2!` is non-allocating and is using the pre-allocated `C` variable directly.

Another example of this is solutions to linear equations, where for large solutions you may pre-allocate and reuse the solution vector.

In [57]:
`A = rand(10,10)
   y = rand(10)
z = A \ y  # creates temporary`    

`A = factorize(A)  # in-place requires factorization
x = similar(y)    # pre-allocate
ldiv!(x, A, y)    # in-place left divide, using factorization`

Out[57]:
10-element Array{Float64,1}:
-0.09745394360765254 0.7799354221131604 1.199436228906085 0.0913844576787099
-0.5083914639425638 -0.3509162355608617 0.793473601987608 -0.5304171009174155
0.451744530913852 -0.8005334538688558

However, if you benchmark carefully, you will see that this is sometimes slower. Avoiding allocations is not always a good idea - and worrying about it prior to benchmarking is premature optimization.

There are a variety of other non-allocating versions of functions. For example,

In [58]:
`A = rand(10,10)
   B = similar(A)`

`transpose!(B, A)  # non-allocating version of B = transpose(A)`

Out[58]:
10×10 Array{Float64,2}:
0.373481  0.715094  0.880197   ...  0.903144  0.0534784  0.646242
0.0572854 0.437244  0.0465054  0.271735  0.0419775  0.91462  0.0572854
0.0722476 0.435665  0.631825   ...  0.773098  0.254097  0.674881
0.0341739 0.185395  0.736277  0.142816  0.687287  0.236726  0.19037
0.843743  0.860459  0.709686  0.630887  0.274137  0.958363  0.948974
0.918731  0.933097  0.280531   ...  0.301123  0.486534  0.988241

Finally, a common source of unnecessary allocations is when taking slices or portions of matrices. For example, the following allocates a new matrix \( B \) and copies the values.

```plaintext
In [59]: A = rand(5,5)
   B = A[2,:]  # extract a vector

Out[59]: 5-element Array{Float64,1}:
  0.07265755245781103
  0.2967203620355736
  0.7745398448673058
  0.624448536072318
  0.5287113274542306
```

To see that these are different matrices, note that

```plaintext
In [60]: A[2,1] = 100.0
  @show A[2,1]
  @show B[1];

A[2, 1] = 100.0
B[1] = 0.07265755245781103
```

Instead of allocating a new matrix, you can take a `view` of a matrix, which provides an appropriate `AbstractArray` type that doesn’t allocate new memory with the `@view` matrix.

```plaintext
In [61]: A = rand(5,5)
   B = @view A[2,:]  # does not copy the data

A[2,1] = 100.0
  @show A[2,1]
  @show B[1];

A[2, 1] = 100.0
B[1] = 100.0
```

But again, you will often find that doing `@view` leads to slower code. Benchmark instead, and generally rely on it for large matrices and for contiguous chunks of memory (e.g., columns rather than rows).

### 22.7 Exercises

#### 22.7.1 Exercise 1

This exercise is for practice on writing low-level routines (i.e., “kernels”), and to hopefully convince you to leave low-level code to the experts.

The formula for matrix multiplication is deceptively simple. For example, with the product of square matrices \( C = AB \) of size \( N \times N \), the \( i,j \) element of \( C \) is
\[ C_{ij} = \sum_{k=1}^{N} A_{ik} B_{kj} \]

Alternatively, you can take a row \( A_{i,:} \) and column \( B_{:,j} \) and use an inner product

\[ C_{ij} = A_{i,:} \cdot B_{:,j} \]

Note that the inner product in a discrete space is simply a sum, and has the same complexity as the sum (i.e., \( O(N) \) operations).

For a dense matrix without any structure and using a naive multiplication algorithm, this also makes it clear why the complexity is \( O(N^3) \): You need to evaluate it for \( N^2 \) elements in the matrix and do an \( O(N) \) operation each time.

For this exercise, implement matrix multiplication yourself and compare performance in a few permutations.

1. Use the built-in function in Julia (i.e., \( C = A \ast B \), or, for a better comparison, the in-place version \( \text{mul!}(C, A, B) \), which works with pre-allocated data).

2. Loop over each \( C_{ij} \) by the row first (i.e., the \( i \) index) and use a \texttt{for} loop for the inner product.

3. Loop over each \( C_{ij} \) by the column first (i.e., the \( j \) index) and use a \texttt{for} loop for the inner product.

4. Do the same but use the \texttt{dot} product instead of the sum.

5. Choose your best implementation of these, and then for matrices of a few different sizes (\( N=10, N=1000 \), etc.), and compare the ratio of performance of your best implementation to the built-in BLAS library.

A few more hints:

- You can just use random matrices (e.g., \( A = \text{rand}(N, N) \)).
- For all of them, pre-allocate the \( C \) matrix beforehand with \( C = \text{similar}(A) \) or something equivalent.
- To compare performance, put your code in a function and use the \texttt{@btime} macro to time it.

### 22.7.2 Exercise 2a

Here we will calculate the evolution of the pdf of a discrete-time Markov chain, \( \psi_t \), given the initial condition \( \psi_0 \).

Start with a simple symmetric tridiagonal matrix

\[
\begin{bmatrix}
0.1 & 0 & \ldots & 0 \\
0 & 0.2 & & \ldots \\
\vdots & \ddots & \ddots & \ddots \\
0 & \ldots & 0.8 & 0.1
\end{bmatrix}
\]

1. Pick some large \( T \) and use the initial condition \( \psi_0 = [1 \ 0 \ \ldots \ 0] \)
2. Write code to calculate $\psi_t$ to some $T$ by iterating the map for each $t$, i.e.,

$$\psi_{t+1} = A' \psi_t$$

1. What is the computational order of calculating $\psi_T$ using this iteration approach $T < N$?

2. What is the computational order of $(A')^T = (A' ... A')$ and then $\psi_T = (A')^T \psi_0$ for $T < N$?

3. Benchmark calculating $\psi_T$ with the iterative calculation above as well as the direct $\psi_T = (A')^T \psi_0$ to see which is faster. You can take the matrix power with just `A_adjoint^T`, which uses specialized algorithms faster and more accurately than repeated matrix multiplication (but with the same computational order).

4. Check the same if $T = 2N$

*Note:* The algorithm used in Julia to take matrix powers depends on the matrix structure, as always. In the symmetric case, it can use an eigendecomposition, whereas with a general dense matrix it uses squaring and scaling.

### 22.7.3 Exercise 2b

With the same setup as in Exercise 2a, do an eigendecomposition of `A_transpose`. That is, use `eigen` to factor the adjoint $A' = Q \Lambda Q^{-1}$, where $Q$ is the matrix of eigenvectors and $A$ is the diagonal matrix of eigenvalues. Calculate $Q^{-1}$ from the results.

Use the factored matrix to calculate the sequence of $\psi_t = (A')^t \psi_0$ using the relationship

$$\psi_t = Q \Lambda^t Q^{-1} \psi_0$$

where matrix powers of diagonal matrices are simply the element-wise power of each element.

Benchmark the speed of calculating the sequence of $\psi_t$ up to $T = 2N$ using this method. In principle, the factorization and easy calculation of the power should give you benefits, compared to simply iterating the map as we did in Exercise 2a. Explain why it does or does not, using computational order of each approach.
Chapter 23

Krylov Methods and Matrix Conditioning

23.1 Contents

- Overview 23.2
- Ill-Conditioned Matrices 23.3
- Stationary Iterative Algorithms for Linear Systems 23.4
- Krylov Methods 23.5
- Iterative Methods for Linear Least Squares 23.6
- Iterative Methods for Eigensystems 23.7
- Krylov Methods for Markov-Chain Dynamics 23.8

23.2 Overview

This lecture takes the structure of numerical methods for linear algebra and builds further toward working with large, sparse matrices. In the process, we will examine foundational numerical analysis such as ill-conditioned matrices.

23.2.1 Setup

In [1]: using InstantiateFromURL

  # optionally add arguments to force installation: instantiate = true,
  precompile = true
  github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

In [2]: using LinearAlgebra, Statistics, BenchmarkTools, Random

  Random.seed!(42);  # seed random numbers for reproducibility

23.2.2 Applications

In this section, we will consider variations on classic problems

1. Solving a linear system for a square $A$ where we will maintain throughout that there is a unique solution to
1. Linear least-squares solution, for a rectangular $A$

$$\min_x \|Ax - b\|^2$$

From theory, we know that if $A$ has linearly independent columns, then the solution is the normal equation

$$x = (A' A)^{-1} A' b$$

1. In the case of a square matrix $A$, the eigenvalue problem is that of finding $x$ and $\lambda$ such that

$$Ax = \lambda x$$

For eigenvalue problems, keep in mind that you do not always require all of the $\lambda$, and sometimes the largest (or smallest) would be enough. For example, calculating the spectral radius requires only the eigenvalue with maximum absolute value.

### 23.3 Ill-Conditioned Matrices

An important consideration in numerical linear algebra, and iterative methods in general, is the condition number.

An ill-conditioned matrix is one where the basis eigenvectors are close to, but not exactly, collinear. While this poses no problem on pen and paper, or with infinite-precision numerical methods, it is important in practice, for two reasons:

1. Ill-conditioned matrices introduce numerical errors roughly in proportion to the base-10 log of the condition number.

2. The convergence speed of many iterative methods is based on the spectral properties of the matrices (e.g., the basis formed by the eigenvectors), and hence ill-conditioned systems can converge slowly.

The solutions to these problems are to

- be careful with operations which introduce error based on the condition number (e.g., matrix inversions when the condition number is high)
- choose, where possible, alternative representations which have less collinearity (e.g., an orthogonal polynomial basis rather than a monomial one)
- use a preconditioner for iterative methods, which changes the spectral properties to increase convergence speed

### 23.3.1 Condition Number

First, let's define and explore the condition number $\kappa$

$$\kappa(A) \equiv \|A\| \|A^{-1}\|$$
where you can use the Cauchy–Schwarz inequality to show that $\kappa(A) \geq 1$. While the condition number can be calculated with any norm, we will focus on the 2-norm.

First, a warning on calculations: Calculating the condition number for a matrix can be an expensive operation (as would calculating a determinant) and should be thought of as roughly equivalent to doing an eigendecomposition. So use it for detective work judiciously.

Let’s look at the condition number of a few matrices using the `cond` function (which allows a choice of the norm, but we’ll stick with the default 2-norm).

```plaintext
In [3]: A = I(2)
    cond(A)
Out[3]: 1.0
```

Here we see an example of the best-conditioned matrix, the identity matrix with its completely orthonormal basis, which has a condition number of 1.

On the other hand, notice that

```plaintext
In [4]: ϵ = 1E-6
       A = [1.0 0.0
            1.0 ϵ]
    cond(A)
Out[4]: 2.0000000000005004e6
```

has a condition number of order $10^6$ - and hence (taking the base-10 log) you would expect to be introducing numerical errors of about 6 significant digits if you are not careful. For example, note that the inverse has both extremely large and extremely small negative numbers

```plaintext
In [5]: inv(A)
Out[5]: 2×2 Array{Float64,2}:
    1.0  0.0
    -1.0e6 1.0e6
```

Since we know that the determinant of nearly collinear matrices is close to zero, this shows another symptom of poor conditioning

```plaintext
In [6]: det(A)
Out[6]: 1.0e-6
```

However, be careful since the determinant has a scale, while the condition number is dimensionless. That is,

```plaintext
In [7]: @show det(1000 * A)
        @show cond(1000 * A);
    det(1000A) = 1.0
    cond(1000A) = 2.0000000000005001e6
```
In that case, the determinant of $A$ is 1, while the condition number is unchanged. This example also provides some intuition that ill-conditioned matrices typically occur when a matrix has radically different scales (e.g., contains both 1 and $1\text{E-6}$, or 1000 and $1\text{E-3}$). This can occur frequently with both function approximation and linear least squares.

### 23.3.2 Condition Numbers and Matrix Operations

Multiplying a matrix by a constant does not change the condition number. What about other operations?

For this example, we see that the inverse has the same condition number (though this will not always be the case).

```python
In [8]: @show cond(A)
    @show cond(inv(A));

    cond(A) = 2.0000000000005004e6
    cond(inv(A)) = 2.0000000002463197e6
```

The condition number of the product of two matrices can change radically and lead things to becoming even more ill-conditioned.

This comes up frequently when calculating the product of a matrix and its transpose (e.g., forming the covariance matrix). A classic example is the Läuchli matrix.

```python
In [9]: lauchli(N, ϵ) = [ones(N)'; ϵ * I(N)]'
ε = 1E-8
L = lauchli(3, ϵ) |> Matrix

Out[9]: 3×4 Array{Float64,2}:
    1.0 1.0e-8 0.0 0.0
    1.0 0.0 1.0e-8 0.0
    1.0 0.0 0.0 1.0e-8
```

Note that the condition number increases substantially

```python
In [10]: @show cond(L)
    @show cond(L' * L);

    cond(L) = 1.732050807568878e8
    cond(L' * L) = 5.345191558726545e32
```

You can show that the analytic eigenvalues of this are $\{3+\epsilon^2, \epsilon^2, \epsilon^2\}$ but the poor conditioning means it is difficult to distinguish these from 0.

This comes up when conducting Principal Component Analysis, which requires calculations of the eigenvalues of the covariance matrix.

```python
In [11]: sort(sqrt.(Complex.(eigen(L*L').values)), lt = (x,y) -> abs(x) < abs(y))
```
Out[11]: 3-element Array{Complex{Float64},1}:
    0.0 + 4.870456104375987e-9im
    4.2146848510894035e-8 + 0.0im
    1.7320508075688772 + 0.0im

Note that these are significantly different than the known analytic solution and, in particular, are difficult to distinguish from 0.

In [12]: sqrt.([3 + ϵ^2, ϵ^2, ϵ^2]) |> sort

Out[12]: 3-element Array{Float64,1}:
    1.0e-8
    1.0e-8
    1.7320508075688772

Alternatively, we could calculate these by taking the square of the singular values of \( L \) itself, which is much more accurate and lets us clearly distinguish from zero

In [13]: svd(L).S |> sort

Out[13]: 3-element Array{Float64,1}:
    9.999999999999997e-9
    1.0e-8
    1.7320508075688774

Similarly, we are better off calculating least squares directly rather than forming the normal equation (i.e., \( A'Ax = A'b \)) ourselves

In [14]: N = 3
A = lauchli(N, 1E-7)' |> Matrix
b = rand(N+1)
x_sol_1 = A \ b  # using a least-squares solver
x_sol_2 = (A' * A) \ (A' * b)  # forming the normal equation ourselves
norm(x_sol_1 - x_sol_2)

Out[14]: 2502.05373776057

23.3.3 Why a Monomial Basis Is a Bad Idea

A classic example of poorly conditioned matrices is using a monomial basis of a polynomial with interpolation.

Take a grid of points, \( x_0, \ldots, x_N \) and values \( y_0, \ldots, y_N \) where we want to calculate the interpolating polynomial.

If we were to use the simplest, and most obvious, polynomial basis, then the calculation consists of finding the coefficients \( c_1, \ldots, c_n \) where

\[
P(x) = \sum_{i=0}^{N} c_i x^i
\]
To solve for the coefficients, we notice that this is a simple system of equations

\[
\begin{align*}
  y_0 &= c_0 + c_1 x_0 + \ldots + c_N x_0^N \\
  \vdots \\
  y_N &= c_0 + c_1 x_N + \ldots + c_N x_N^N
\end{align*}
\]

Or, stacking \(c = [c_0 \ldots c_N], y = [y_0 \ldots y_N]\) and

\[
A = \begin{bmatrix}
  1 & x_0 & x_0^2 & \ldots & x_0^N \\
  \vdots & \vdots & \vdots & \ddots & \vdots \\
  1 & x_N & x_N^2 & \ldots & x_N^N
\end{bmatrix}
\]

We can then calculate the interpolating coefficients as the solution to

\[
Ac = y
\]

Implementing this for the interpolation of the \(exp(x)\) function

```
In [15]: N = 5
f(x) = exp(x)
x = range(0.0, 10.0, length = N+1)
y = f.(x)  # generate some data to interpolate
A = [x_i^n for x_i in x, n in 0:N]
A_inv = inv(A)
c = A_inv * y
norm(A * c - f.(x), Inf)
```

```
Out[15]: 1.356966095045209e-9
```

The final step just checks the interpolation vs. the analytic function at the nodes. Keep in mind that this should be very close to zero since we are interpolating the function precisely at those nodes. In our example, the Inf-norm (i.e., maximum difference) of the interpolation errors at the nodes is around \(1E-9\), which is reasonable for many problems.

But note that with \(N = 5\) the condition number is already of order \(1E6\).

```
In [16]: cond(A)
```

```
Out[16]: 564652.3214053963
```

What if we increase the degree of the polynomial with the hope of increasing the precision of the interpolation?

```
In [17]: N = 10
f(x) = exp(x)
x = range(0.0, 10.0, length = N+1)
y = f.(x)  # generate some data to interpolate
A = [x_i^n for x_i in x, n in 0:N]
A_inv = inv(A)
c = A_inv * y
norm(A * c - f.(x), Inf)
```

```
Out[17]: 1.356966095045209e-9
```

```
Out[16]: 564652.3214053963
```

But note that with \(N = 10\) the condition number is already of order \(1E6\).
Here, we see that hoping to increase the precision between points by adding extra polynomial terms is backfiring. By going to a 10th-order polynomial, we have introduced an error of about $1 \times 10^{-5}$, even at the interpolation points themselves.

This blows up quickly

```
In [18]: N = 20
   f(x) = exp(x)
   x = range(0.0, 10.0, length = N+1)
   y = f.(x)  # generate some data to interpolate

   A = [x_i^n for x_i in x, n in 0:N]
   A_inv = inv(A)
   c = A_inv * y
   norm(A * c - f.(x), Inf)
```

```
Out[18]: 19978.410967681375
```

To see the source of the problem, note that the condition number is astronomical.

```
In [19]: cond(A)
```

```
Out[19]: 2.0386741019186427e24
```

At this point, you should be suspicious of the use of `inv(A)`, since we have considered solving linear systems by taking the inverse as verboten. Indeed, this made things much worse. The error drops dramatically if we solve it as a linear system

```
In [20]: c = A \ y
   norm(A * c - f.(x), Inf)
```

```
Out[20]: 1.864464138634503e-10
```

But an error of $1 \times 10^{-10}$ at the interpolating nodes themselves can be a problem in many applications, and if you increase $N$ then the error will become non-trivial eventually - even without taking the inverse.

The heart of the issue is that the monomial basis leads to a Vandermonde matrix, which is especially ill-conditioned.

### Aside on Runge’s Phenomenon

The monomial basis is also a good opportunity to look at a separate type of error due to Runge’s Phenomenon. It is an important issue in approximation theory, albeit not one driven by numerical approximation errors.

It turns out that using a uniform grid of points is, in general, the worst possible choice of interpolation nodes for a polynomial approximation. This phenomenon can be seen with the interpolation of the seemingly innocuous Runge’s function, $g(x) = \frac{1}{1 + 25x^2}$. 
Let’s calculate the interpolation with a monomial basis to find the $c_i$ such that

$$
\frac{1}{1 + 25x^2} \approx \sum_{i=0}^{N} c_i x^i, \text{ for } -1 \leq x \leq 1
$$

First, interpolate with $N = 5$ and avoid taking the inverse. In that case, as long as we avoid taking an inverse, the numerical errors from the ill-conditioned matrix are manageable.

In [21]: using Plots
   gr(fmt=:png);

   N_display = 100
   g(x) = 1/(1 + 25x^2)
   x_display = range(-1, 1, length = N_display)
   y_display = g.(x_display)

   # interpolation
   N = 5
   x = range(-1.0, 1.0, length = N+1)
   y = g.(x)
   A_5 = [x_i^n for x_i in x, n in 0:N]
   c_5 = A_5 \ y

   # use the coefficients to evaluate on x_display grid
   B_5 = [x_i^n for x_i in x_display, n in 0:N]  # calculate monomials for display grid
   y_5 = B_5 * c_5  # calculates for each in x_display_grid
   plot(x_display, y_5, label = "P_5(x)")
   plot!(x_display, y_display, w = 3, label = "g(x)")
Note that while the function, \( g(x) \), and the approximation with a 5th-order polynomial, \( P_5(x) \), coincide at the 6 nodes, the approximation has a great deal of error everywhere else.

The oscillations near the boundaries are the hallmarks of Runge’s Phenomenon. You might guess that increasing the number of grid points and the order of the polynomial will lead to better approximations:

In [22]:

```python
N = 9
x = range(-1.0, 1.0, length = N+1)
y = g(x)
A_9 = [x_i^n for x_i in x, n in 0:N]
c_9 = A_9 \ y

# use the coefficients to evaluate on x_display grid
B_9 = [x_i^n for x_i in x_display, n in 0:N]  # calculate monomials for display grid
y_9 = B_9 * c_9  # calculates for each in x_display_grid
plot(x_display, y_9, label = "P_9(x)")
plot!(x_display, y_display, w = 3, label = "g(x)")
```

Out[22]:

While the approximation is better near \( x=0 \), the oscillations near the boundaries have become worse. Adding on extra polynomial terms will not globally increase the quality of the approximation.

**Using an Orthogonal Polynomial Basis**

We can minimize the numerical problems of an ill-conditioned basis matrix by choosing a different basis for the polynomials.
For example, Chebyshev polynomials form an orthonormal basis under an appropriate inner product, and we can form precise high-order approximations, with very little numerical error.

In [23]: using ApproxFun
N = 10000
S = Chebyshev(-1.0..1.0) # form Chebyshev basis
x = points(S, N) # choose Chebyshev nodes
y = g(x)
g_approx = Fun(S, ApproxFun.transform(S,y)) # transform fits the polynomial
@show norm(g_approx.(x) - g.(x), Inf)

Out[23]:

Besides the use of a different polynomial basis, we are approximating at different nodes (i.e., Chebyshev nodes). Interpolation with Chebyshev polynomials at the Chebyshev nodes ends up minimizing (but not eliminating) Runge’s Phenomenon.

Lessons for Approximation and Interpolation

To summarize:

1. Check the condition number on systems you suspect might be ill-conditioned (based on intuition of collinearity).
2. If you are working with ill-conditioned matrices, be especially careful not to take the
inverses or multiply by the transpose.

3. Avoid a monomial polynomial basis. Instead, use polynomials (e.g., Chebyshev or La-
grange) orthogonal under an appropriate inner product, or use a non-global basis such
as cubic splines.

4. If possible, avoid using a uniform grid for interpolation and approximation, and choose
nodes appropriate for the basis.

However, sometimes you can’t avoid ill-conditioned matrices. This is especially common with
discretization of PDEs and with linear least squares.

### 23.4 Stationary Iterative Algorithms for Linear Systems

As before, consider solving the equation

\[ Ax = b \]

We will now focus on cases where \( A \) is both massive (e.g., potentially millions of equations)
and sparse, and sometimes ill-conditioned - but where there is always a unique solution.

While this may seem excessive, it occurs in practice due to the curse of dimensionality, dis-
cretizations of PDEs, and when working with big data.

The methods in the previous lectures (e.g., factorization and approaches similar to Gaussian
elimination) are called direct methods, and are able in theory to converge to the exact solu-
tion in a finite number of steps while directly working with the matrix in memory.

Instead, iterative solutions start with a guess on a solution and iterate until convergence. The
benefit will be that each iteration uses a lower-order operation (e.g., an \( O(N^2) \) matrix-vector
product) which will make it possible to

1. solve much larger systems, even if done less precisely.
2. define linear operators in terms of the matrix-vector products, rather than storing as a
matrix.
3. get approximate solutions in progress prior to the completion of all algorithm steps, un-
like the direct methods, which provide a solution only at the end.

Of course, there is no free lunch, and the computational order of the iterations themselves
would be comparable to the direct methods for a given level of tolerance (e.g., \( O(N^3) \) opera-
tions may be required to solve a dense unstructured system).

There are two types of iterative methods we will consider. The first type is stationary meth-
ods, which iterate on a map in a way that’s similar to fixed-point problems, and the second
type is Krylov methods, which iteratively solve using left-multiplications of the linear opera-
tor.

For our main examples, we will use the valuation of the continuous-time Markov chain from
the numerical methods for linear algebra lecture. That is, given a payoff vector \( r \), a discount
rate \( \rho \), and the infinitesimal generator of the Markov chain \( Q \), solve the equation
\[ \rho v = r + Qv \]

With the sizes and types of matrices here, iterative methods are inappropriate in practice, but they will help us understand the characteristics of convergence and how they relate to matrix conditioning.

### 23.4.1 Stationary Methods

First, we will solve with a direct method, which will give the solution to machine precision.

In [24]: using LinearAlgebra, IterativeSolvers, Statistics

\[
\begin{align*}
\alpha &= 0.1 \\
N &= 100 \\
Q &= \text{Tridiagonal}(\text{fill}(\alpha, N-1), [-\alpha; \text{fill}(-2\alpha, N-2); -\alpha], \text{fill}(\alpha, N-1)) \\
r &= \text{range}(0.0, 10.0, \text{length}=N) \\
\rho &= 0.05 \\
A &= \rho \cdot I - Q \\
v\_direct &= A \setminus r \\
\text{mean}(v\_direct)
\end{align*}
\]

Out[24]: 100.00000000000004

Without proof, consider that given the discount rate of \( \rho > 0 \), this problem could be set up as a contraction for solving the Bellman equation through methods such as value-function iteration.

The condition we will examine here is called **diagonal dominance**.

\[ |A_{ii}| \geq \sum_{j \neq i} |A_{ij}| \quad \text{for all } i = 1 \ldots N \]

That is, in every row, the diagonal element is weakly greater in absolute value than the sum of all of the other elements in the row. In cases where it is strictly greater, we say that the matrix is strictly diagonally dominant.

With our example, given that \( Q \) is the infinitesimal generator of a Markov chain, we know that each row sums to 0, and hence it is weakly diagonally dominant.

However, notice that when \( \rho > 0 \), and since the diagonal of \( Q \) is negative, \( A = \rho I - Q \) makes the matrix strictly diagonally dominant.

### 23.4.2 Jacobi Iteration

For matrices that are **strictly diagonally dominant**, you can prove that a simple decomposition and iteration procedure will converge.

To solve a system \( Ax = b \), split the matrix \( A \) into its diagonal and off-diagonal elements.

That is,

\[ A = D + R \]
where

\[
D = \begin{bmatrix}
A_{11} & 0 & \ldots & 0 \\
0 & A_{22} & \ldots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
0 & 0 & \ldots & A_{NN}
\end{bmatrix}
\]

and

\[
R = \begin{bmatrix}
0 & A_{12} & \ldots & A_{1N} \\
A_{21} & 0 & \ldots & A_{2N} \\
\vdots & \vdots & \ddots & \vdots \\
A_{N1} & A_{N2} & \ldots & 0
\end{bmatrix}
\]

Rearrange the \((D + R)x = b\) as

\[
Dx = b - Rx \\
x = D^{-1}(b - Rx)
\]

where, since \(D\) is diagonal, its inverse is trivial to calculate with \(O(N)\) complexity.

To solve, take an iteration \(x^k\), starting from \(x^0\), and then form a new guess with

\[
x^{k+1} = D^{-1}(b - Rx^k)
\]

The complexity here is \(O(N^2)\) for the matrix-vector product, and \(O(N)\) for the vector subtraction and division.

The package \texttt{IterativeSolvers.jl} package implements this method.

For our example, we start with a guess and solve for the value function and iterate

\begin{verbatim}
In [25]: using IterativeSolvers, LinearAlgebra, SparseArrays
   v = zeros(N)
   jacobi!(v, A, r, maxiter = 40)
  @show norm(v - v_direct, Inf)
   norm(v - v_direct, Inf) = 0.022858373200932647

Out[25]: 0.022858373200932647
\end{verbatim}

With this, after 40 iterations we see that the error is in the order of \(1E-2\)

### 23.4.3 Other Stationary Methods

In practice, there are many methods that are better than Jacobi iteration. For example \texttt{Gauss-Siedel}, which splits the matrix \(A = L + U\) into a lower-triangular matrix \(L\) and an upper-triangular matrix \(U\) without the diagonal.

The iteration becomes
\[ Lx^{k+1} = b - Ux^k \]

In that case, since the \( L \) matrix is triangular, the system can be solved in \( O(N^2) \) operations after \( b - Ux^k \) is formed.

In [26]:
```julia
v = zeros(N)
gauss_seidel!(v, A, r, maxiter = 40)
@show norm(v - v_direct, Inf);
```
```
norm(v - v_direct, Inf) = 1.5616376089155892e-5
```

The accuracy increases substantially. After 40 iterations, we see that the error is of the order of \( 1E-5 \).

Another example is Successive Over-relaxation (SOR), which takes a relaxation parameter \( \omega > 1 \) and decomposes the matrix as \( A = L + D + U \), where \( L, U \) are strictly upper- and lower-diagonal matrices and \( D \) is diagonal.

Decompose the \( A \) matrix, multiply the system by \( \omega \), and rearrange to find

\[ (D + \omega L)x^{k+1} = \omega b - (\omega U + (\omega - 1)D)x^k \]

In that case, \( D + \omega L \) is a triangular matrix, and hence the linear solution is \( O(N^2) \).

In [27]:
```julia
v = zeros(N)
sor!(v, A, r, 1.1, maxiter = 40)
@show norm(v - v_direct, Inf);
```
```
norm(v - v_direct, Inf) = 3.745356593753968e-7
```

The accuracy is now \( 1E-7 \). If we change the parameter to \( \omega = 1.2 \), the accuracy further increases to \( 1E-9 \).

This technique is common with iterative methods: Frequently, adding a damping or a relaxation parameter will counterintuitively speed up the convergence process.

**Note:** The stationary iterative methods are not always used directly, but are sometimes used as a “smoothing” step (e.g., running 5-10 times) prior to using other Krylov methods.

### 23.5 Krylov Methods

A more commonly used set of iterative methods is based on Krylov subspaces, which involve iterating the \( A^k x \) matrix-vector product, and orthogonalizing to ensure that the resulting iteration is not too collinear.

The prototypical Krylov method is Conjugate Gradient, which requires the \( A \) matrix to be symmetric and positive definite.

Solving an example:
23.5. KRYLOV METHODS

In [28]: N = 100
   A = sprand(100, 100, 0.1)  # 10 percent non-zeros
   A = A * A'  # easy way to generate a symmetric positive-definite matrix
   @show isposdef(A)
   b = rand(N)
   x_direct = A \ b  # sparse direct solver more appropriate here
   cond(Matrix(A * A'))

   isposdef(A) = true

Out[28]: 3.5791585364800934e10

Notice that the condition numbers tend to be large for large random matrices.

Solving this system with the conjugate gradient method:

In [29]: x = zeros(N)
   sol = cg!(x, A, b, log=true, maxiter = 1000)
   sol[end]

Out[29]: Converged after 174 iterations.

23.5.1 Introduction to Preconditioning

If you tell a numerical analyst that you are using direct methods, their first question may be, “which factorization?” But if you tell them you are using an iterative method, they may ask “which preconditioner?”.

As discussed at the beginning of the lecture, the spectral properties of matrices determine the rate of convergence of iterative methods. In particular, ill-conditioned matrices can converge slowly with iterative methods, for the same reasons that naive value-function iteration will converge slowly if the discount rate is close to 1.

Preconditioning solves this problem by adjusting the spectral properties of the matrix, at the cost of some extra computational operations.

To see an example of a right-preconditioner, consider a matrix $P$ which has a convenient and numerically stable inverse. Then

$$Ax = b$$
$$AP^{-1}P_x = b$$
$$AP^{-1}y = b$$
$$Px = y$$

That is, solve $(AP^{-1})y = b$ for $y$, and then solve $Px = y$ for $x$.

There are all sorts of preconditioners, and they are specific to the particular problem at hand. The key features are that they have convenient (and lower-order!) ways to solve the resulting system and they lower the condition number of the matrix. To see this in action, we can look at a simple preconditioner.

The diagonal precondition is simply $P = \text{Diagonal}(A)$. Depending on the matrix, this can change the condition number a little or a lot.
In [30]: AP = A * inv(Diagonal(A))
@show cond(Matrix(A))
@show cond(Matrix(AP));

cond(Matrix(A)) = 189186.6473381337
cond(Matrix(AP)) = 175174.59095330362

But it may or may not decrease the number of iterations

In [31]: using Preconditioners
x = zeros(N)
P = DiagonalPreconditioner(A)
sol = cg!(x, A, b, log=true, maxiter = 1000)
sol[end]
Out[31]: Converged after 174 iterations.

Another classic preconditioner is the incomplete LU decomposition

In [32]: using IncompleteLU
x = zeros(N)
P = ilu(A, \tau = 0.1)
sol = cg!(x, A, b, Pl = P, log=true, maxiter = 1000)
sol[end]
Out[32]: Converged after 86 iterations.

The \tau parameter determines the degree of the LU decomposition to conduct, providing a tradeoff between preconditioner and solve speed.

A good rule of thumb is that you should almost always be using a preconditioner with iterative methods, and you should experiment to find preconditioners that are appropriate for your problem.

Finally, naively trying another preconditioning approach (called Algebraic Multigrid) gives us a further drop in the number of iterations.

In [33]: x = zeros(N)
P = AMGPreconditioner{RugeStuben}(A)
sol = cg!(x, A, b, Pl = P, log=true, maxiter = 1000)
sol[end]
Out[33]: Converged after 59 iterations.

*Note: Preconditioning is also available for stationary, iterative methods (see this example), but is frequently not implemented since such methods are not often used for the complete solution.*
23.5. **KRYLOV METHODS**

### 23.5.2 Methods for General Matrices

There are many algorithms which exploit matrix structure (e.g., the conjugate gradient method for positive-definite matrices, and MINRES for matrices that are only symmetric/Hermitian).

On the other hand, if there is no structure to a sparse matrix, then GMRES is a good approach.

To experiment with these methods, we will use our ill-conditioned interpolation problem with a monomial basis.

**In [34]:** using IterativeSolvers

```julia
N = 10
f(x) = exp(x)
x = range(0.0, 10.0, length = N+1)
y = f(x)  # generate some data to interpolate
A = sparse([x_i^n for x_i in x, n in 0:N])
c = zeros(N+1)  # initial guess required for iterative solutions
results = gmres!(c, A, y, log=true, maxiter = 1000)
println("cond(A) = \$\$(cond(Matrix(A))), \$(results[end]) Norm error\n-\$(norm(A*c - y, Inf))")
```

```
cond(A) = 4.462833495403007e12, Converged after 11 iterations. Norm error 7.62520357966423e-8
```

That method converged in 11 iterations. Now if we try it with an incomplete LU preconditioner, we see that it converges immediately.

**In [35]:** N = 10

```julia
f(x) = exp(x)
x = range(0.0, 10.0, length = N+1)
y = f(x)  # generate some data to interpolate
A = [x_i^n for x_i in x, n in 0:N]
P = ilu(sparse(A), τ = 0.1)
c = zeros(N+1)  # initial guess required for iterative solutions
results = gmres!(c, A, y, P1 = P, log=true, maxiter = 1000)
println("\$(results[end]) Norm error \$(norm(A*c - y, Inf))")
```

```
Converged after 1 iterations. Norm error 4.5034175855107605e-7
```

With other preconditioners (e.g., DiagonalPreconditioner), we may save only one or two iterations. Keep in mind, however, to consider the cost of the preconditioning process in your problem.

### 23.5.3 Matrix-Free Methods

First, let's use a Krylov method to solve our simple valuation problem...
In [36]: α = 0.1
N = 100
Q = Tridiagonal(fill(α, N-1), [-α; fill(-2α, N-2); -α], fill(α, N-1))

r = range(0.0, 10.0, length=N)
ψ = 0.05
A = ψ * I - Q
v = zeros(N)
results = gmres!(v, A, r, log=true)
v_sol = results[1]
println("\$\pounds(results[end])")

Converged after 20 iterations.

While the $A$ matrix was important to be kept in memory for direct methods, Krylov methods such as GMRES are built on matrix-vector products, i.e., $Ax$ for iterations on the $x$.

This product can be written directly for a given $x$,

$$
Ax = \begin{bmatrix}
(r + \alpha)x_1 - \alpha x_2 \\
-\alpha x_1 + (r + 2\alpha)x_2 - \alpha x_3 \\
\vdots \\
-\alpha x_{N-2} + (r + 2\alpha)x_{N-1} - \alpha x_N \\
-\alpha x_{N-1} + (r + \alpha)x_N
\end{bmatrix}
$$

This can be implemented as a function (either in-place or out-of-place) which calculates $y = Ax$

In [37]: A_mul(x) = [ (ψ + α) * x[1] - α * x[2];
[-α * x[i-1] + (ψ + 2*α) * x[i] - α * x[i+1] for i in 2:N-1]; #
comprehension
- α * x[end-1] + (ψ + α) * x[end]]

x = rand(N)
@show norm(A * x - A_mul(x))  # compare to matrix;

norm(A * x - A_mul(x)) = 0.0

The final line verifies that the $A$ _mul_ function provides the same result as the matrix multiplication with our original $A$ for a random vector.

In abstract mathematics, a finite-dimensional linear operator is a mapping $A: \mathbb{R}^N \rightarrow \mathbb{R}^N$ that satisfies a number of criteria such as $A(c_1x_1 + c_2x_2) = c_1Ax_1 + c_2Ax_2$ for scalars $c_i$ and vectors $x_i$.

Moving from abstract mathematics to generic programming, we can think of a linear operator as a map that satisfies a number of requirements (e.g., it has a left-multiply to apply the map $*$, an in-place left-multiply $mul!$, and an associated size). A Julia matrix is just one possible implementation of the abstract concept of a linear operator.

Convenience wrappers can provide some of the boilerplate which turns the $A$ _mul_ function into something that behaves like a matrix. One package is LinearMaps.jl and another is LinearOperators.jl.
In [38]: using LinearMaps
    A_map = LinearMap(A_mul, N)  # map uses the A_mul function

Out[38]: LinearMaps.FunctionMap{Float64}(A_mul, 100, 100; ismutating=false,
    issymmetric=false, ishermitian=false, isposdef=false)

Now, with the A_map object, we can fulfill many of the operations we would expect from a matrix

In [39]: x = rand(N)
    @show norm(A_map * x - A * x)
    y = similar(x)
    mul!(y, A_map, x)  # in-place multiplication
    @show norm(y - A * x)
    @show size(A_map)
    @show norm(Matrix(A_map) - A)
    @show nnz(sparse(A_map));

    norm(A_map * x - A * x) = 0.0
    norm(y - A * x) = 0.0
    size(A_map) = (100, 100)
    norm(Matrix(A_map) - A) = 0.0
    nnz(sparse(A_map)) = 298

Note: In the case of sparse(A_map) and Matrix(A_map), the code is using the left-
multiplication operator with N standard basis vectors to construct the full matrix. This
should be used only for testing purposes.

But notice that as the linear operator does not have indexing operations, it is not an array or
a matrix.

In [40]: typeof(A_map) <: AbstractArray

Out[40]: false

As long as algorithms using linear operators are written generically (e.g., using the matrix-
vector * or mul! functions) and the types of functions are not unnecessarily constrained to
be Matrix or AbstractArray when it isn’t strictly necessary, then the A_map type can
work in places which would otherwise require a matrix.

For example, the Krylov methods in IterativeSolvers.jl are written for generic left-
multiplication

In [41]: results = gmres(A_map, r, log = true)  # Krylov method using the matrix-
free type
    println("$results[end])")

Converged after 20 iterations.
These methods are typically not competitive with sparse, direct methods unless the problems become very large. In that case, we often want to work with pre-allocated vectors. Instead of using \( y = A \times x \) for matrix-vector products, we would use the in-place \( \text{mul!}(y, A, x) \) function. The wrappers for linear operators all support in-place non-allocating versions for this purpose.

```
In [42]:

    def A_mul!(y, x): # in-place version
        for i in 2:N-1
            y[i] = -α * x[i-1] + (ρ + 2α) * x[i] - α * x[i+1]
        end
        y[end] = -α * x[end-1] + (ρ + α) * x[end]
        return y

A_map_2 = LinearMap(A_mul!, N, ismutating = true) # ismutating == in-place

v = zeros(N)
@show norm(A_map_2 * v - A * v) # can still call with * and have it allocate

results = gmres!(v, A_map_2, r, log=true) # in-place gmres
println("\$\n(results[end])")
```

Finally, keep in mind that the linear operators can compose, so that \( A(c_1 x) + B(c_2 x) + x = (c_1 A + c_2 B + I) x \) is well defined for any linear operators - just as it would be for matrices \( A, B \) and scalars \( c_1, c_2 \).

For example, take \( 2A x + x = (2A + I) x = B x \) as a new linear map, 

```
In [43]: B = 2.0 * A_map + I  # composite linear operator
B * rand(N)  # left-multiply works with the composition
typeof(B)
```

```
Out[43]: LinearMaps.LinearCombination{Float64,Tuple{LinearMaps.
    CompositeMap{Float64,Tuple{LinearMaps.
        FunctionMap{Float64,typeof(A_mul),Nothing},LinearMaps.
        UniformScalingMap{Float64}}},L
        linearMaps.UniformScalingMap{Bool}}}
```

The wrappers, such as \( \text{LinearMap} \) wrappers, make this composition possible by keeping the composition graph of the expression (i.e., \( \text{LinearCombination} \)) and implementing the left-multiply recursively using the rules of linearity.

Another example is to solve the \( \rho v = r + Q v \) equation for \( v \) with composition of matrix-free methods for \( L \) rather than by creating the full \( A = \rho - Q \) operator, which we implemented as \( A \_mul \)

```
In [44]: Q_mul(x) = [ -α * x[1] + α * x[2]; [α * x[i-1] - 2*α * x[i] + α*x[i+1] for i in 2:N-1]; #]
    α * x[end-1] - α * x[end];
```

```
23.6 Iterative Methods for Linear Least Squares

In theory, the solution to the least-squares problem, \( \min_x \|Ax - b\|_2^2 \), is simply the solution to the normal equations \((A'A)x = A'b\).

We saw, however, that in practice, direct methods use a QR decomposition - in part because an ill-conditioned matrix \(A\) becomes even worse when \(A'A\) is formed.

For large problems, we can also consider Krylov methods for solving the linear least-squares problem. One formulation is the LSMR algorithm, which can solve the regularized

\[
\min_x \|Ax - b\|_2^2 + \|\lambda x\|_2^2
\]

The purpose of the \(\lambda \geq 0\) parameter is to dampen the iteration process and/or regularize the solution. This isn’t required, but can help convergence for ill-conditioned matrices \(A\). With the damping parameter, the normalized equations would become \((A'A + \lambda^2 I)x = A'b\).

We can compare solving the least-squares problem with LSMR and direct methods.
implement an iterative solution. Unlike the previous versions, the left-multiply is insufficient since the least squares also deals with the transpose of the operator. For this reason, in order to use matrix-free methods, we need to define the $A \times x$ and $\text{transpose}(A) \times y$ functions separately.

```python
In [46]: # Could implement as matrix-free functions.
    X_func(u) = X * u  # matrix-vector product
    X_T_func(v) = X' * v  # i.e., adjoint-vector product

    X_map = LinearMap(X_func, X_T_func, N, M)
    results = lsmr(X_map, y, log = true)
    println("$\$(results[end])")

    Converged after 14 iterations.
```

### 23.7 Iterative Methods for Eigensystems

When you use `eigen` on a dense matrix, it calculates an eigendecomposition and provides all the eigenvalues and eigenvectors.

While this is sometimes necessary, a spectral decomposition of a dense, unstructured matrix is one of the costliest $O(N^3)$ operations (i.e., it has one of the largest constants). For large matrices, it is often infeasible.

Luckily, we frequently need only a few eigenvectors/eigenvalues (in some cases just one), which enables a different set of algorithms.

For example, in the case of a discrete-time Markov chain, in order to find the stationary distribution, we are looking for the eigenvector associated with the eigenvalue 1. As usual, a little linear algebra goes a long way.

From the Perron-Frobenius theorem, the largest eigenvalue of an irreducible stochastic matrix is 1 - the same eigenvalue we are looking for.

Iterative methods for solving eigensystems allow targeting the smallest magnitude, the largest magnitude, and many others. The easiest library to use is `Arpack.jl`.

As an example,

```python
In [47]: using Arpack, LinearAlgebra
    N = 1000
    A = Tridiagonal([fill(0.1, N-2); 0.2], fill(0.8, N), [0.2; fill(0.1, N-2)];

    A_adjoint = A'

    # Find 1 of the largest magnitude eigenvalue
    λ, ϕ = eigs(A_adjoint, nev=1, which=:LM, maxiter=1000)
    ϕ = real(ϕ) ./ sum(real(ϕ))
   @show λ
   @show mean(ϕ);

    λ = Complex{Float64}[1.0000000000000189 + 0.0im]
    mean(ϕ) = 0.0010000000000002
```
Indeed, the $\lambda$ is equal to 1. If we choose nev = 2, it will provide the eigenpairs with the two eigenvalues of largest absolute value.

**Hint:** If you get errors using Arpack, increase the maxiter parameter for your problems.

Iterative methods for eigensystems rely on matrix-vector products rather than decompositions, and are amenable to matrix-free approaches. For example, take the Markov chain for a simple counting process:

1. The count starts at 1 and has a maximum of $N$.
2. With probability $\theta \geq 0$, an existing count is lost with probability $\zeta \geq 0$ such that $\theta + \zeta \leq 1$.
3. If the count is at 1, then the only transition is to add a count with probability $\theta$.
4. If the current count is $N$, then the only transition is to lose the count with probability $\zeta$.

First, finding the transition matrix $P$ and its adjoint directly as a check

\[
\begin{align*}
\text{In [48]: } & \theta = 0.1 \\
& \zeta = 0.05 \\
& N = 5 \\
& P = \text{Tridiagonal}(\text{fill}(\zeta, N-1), [1-\theta; \text{fill}(1-\theta \cdot \zeta, N-2); 1-\zeta], \text{fill}(\theta, N-1)) \\
\text{P'}
\end{align*}
\]

\[
\begin{align*}
\text{Out[48]: } & 5 \times 5 \text{ Tridiagonal}\{\text{Float64,Array[Float64,1]}\}: \\
& \begin{bmatrix}
0.9 & 0.05 & 0 & 0 & 0 \\
0.1 & 0.85 & 0.05 & 0 & 0 \\
0 & 0.1 & 0.85 & 0.05 & 0 \\
0 & 0 & 0.1 & 0.85 & 0.05 \\
0 & 0 & 0 & 0.1 & 0.95
\end{bmatrix}
\end{align*}
\]

Implementing the adjoint-vector product directly, and verifying that it gives the same matrix as the adjoint

\[
\begin{align*}
\text{In [49]: } & \text{P_adj_mul}(x) = [ (1-\theta) \times x[1] + \zeta \times x[2]; \\
& [\theta \times x[i-1] + (1-\theta \cdot \zeta) \times x[i] + \zeta \times x[i+1] \text{ for i in } 2:N-1]; \#
\end{align*}
\]

\[
\text{comprehension}
\]

\[
\theta \times x[end-1] + (1-\zeta) \times x[end];
\]

\[
\text{P_adj_map} = \text{LinearMap}(\text{P_adj_mul}, N)
\]

\[
@\text{show} \ \text{norm}(P' - \text{sparse}(\text{P_adj_map}))
\]

\[
\text{norm}(P' - \text{sparse}(\text{P_adj_map})) = 0.0
\]

\[
\text{Out[49]: } 0.0
\]

Finally, solving for the stationary distribution using the matrix-free method (which could be verified against the decomposition approach of $P'$)
In [50]: \( \lambda, \phi = \text{eigs(P_adj_map, nev=1, which=LM, maxiter=1000)} \)
\( \phi = \text{real}(\phi) ./ \text{sum(real}(\phi)) \)

\( \lambda = \text{Complex\{Float64\}[1.0 + 0.0im]} \)
\( \phi = [0.03225806451612657; 0.06451612903225695; 0.1290322580645172; \\
0.25806451612903425; 0.516129032258065] \)

Out[50]: 5×1 Array\{Float64,2\}:
0.03225806451612657
0.06451612903225695
0.1290322580645172
0.25806451612903425
0.516129032258065

Of course, for a problem this simple, the direct eigendecomposition will be significantly faster. Use matrix-free iterative methods only for large systems where you do not need all of the eigenvalues.

23.8 Krylov Methods for Markov-Chain Dynamics

This example applies the methods in this lecture to a large continuous-time Markov chain, and provides some practice working with arrays of arbitrary dimensions.

Consider a version of the Markov-chain dynamics in [85], where a firm has a discrete number of customers of different types. To keep things as simple as possible, assume that there are \( m = 1, \ldots, M \) types of customers and that the firm may have \( n = 1, \ldots, N \) customers of each type.

To set the notation, let \( n_m \in \{1, \ldots, N\} \) be the number of customers of type \( m \), so that the state of a firm is \( \{n_1, \ldots, n_m, \ldots, n_M\} \). The cardinality of possible states is then \( \mathbf{N} \equiv N^M \), which can blow up quickly as the number of types increases.

The stochastic process is a simple counting/forgetting process, as follows:

1. For every \( 1 \leq n_m(t) < N \), there is a \( \theta \) intensity of arrival of a new customer, so that \( n_m(t + \Delta) = n_m(t) + 1 \).
2. For every \( 1 < n_m(t) \leq N \), there is a \( \zeta \) intensity of losing a customer, so that \( n_m(t + \Delta) = n_m(t) - 1 \).

23.8.1 Matrix-free Infinitesimal Generator

In order to define an intensity matrix \( Q \) of size \( \mathbf{N} \times \mathbf{N} \), we need to choose a consistent ordering of the states. But before we enumerate them linearly, take a \( \nu \in \mathbb{R}^\mathbf{N} \) interpreted as a multidimensional array and look at the left product of the linear operator \( Q\nu \rightarrow \mathbb{R}^\mathbf{N} \).

For example, if we were implementing the product at the row of \( Q \) corresponding to the \( (n_1, \ldots, n_M) \) state, then
\[ Q_{(n_1,\ldots,n_M)} \cdot v = \theta \sum_{m=1}^{M} (n_m < N)v(n_1,\ldots,n_m + 1,\ldots,n_M) \]
\[ + \zeta \sum_{m=1}^{M} (1 < n_m)v(n_1,\ldots,n_m - 1,\ldots,n_M) \]
\[ - (\theta \text{ Count}(n_m < N) + \zeta \text{ Count}(n_m > 1)) v(n_1,\ldots,n_M) \]

Here:
- the first term includes all of the arrivals of new customers into the various \( m \)
- the second term is the loss of a customer for the various \( m \)
- the last term is the intensity of all exits from this state (i.e., counting the intensity of all other transitions, to ensure that the row will sum to 0)

In practice, rather than working with the \( f \) as a multidimensional type, we will need to enumerate the discrete states linearly, so that we can iterate \( f \) between 1 and \( N \). An especially convenient approach is to enumerate them in the same order as the \( K \)-dimensional Cartesian product of the \( N \) states in the multi-dimensional array above.

This can be done with the \texttt{CartesianIndices} function, which is used internally in Julia for the \texttt{eachindex} function. For example,

In [51]:
N = 2
M = 3
shape = \texttt{Tuple}(fill(N, M))
v = \texttt{rand}(shape...)
@show typeof(v)

for \texttt{ind} in \texttt{CartesianIndices(v)}
    println("v$(\texttt{ind}.I) = $(\texttt{v}[\texttt{ind}])") # .I gets the tuple to display
end

\texttt{typeof(v) = Array\{Float64,3\}}
\texttt{v(1, 1, 1) = 0.639089412234831}
\texttt{v(2, 1, 1) = 0.4302368488000152}
\texttt{v(1, 2, 1) = 0.21490768283644002}
\texttt{v(2, 2, 1) = 0.7542051014748841}
\texttt{v(1, 1, 2) = 0.4330861190374067}
\texttt{v(2, 1, 2) = 0.07556766967902084}
\texttt{v(1, 2, 2) = 0.2143739072351467}
\texttt{v(2, 2, 2) = 0.43231874437572815}

The added benefit of this approach is that it will be the most efficient way to iterate through vectors in the implementation.

For the counting process with arbitrary dimensions, we will frequently be incrementing or decrementing the \( m \) unit vectors of the \texttt{CartesianIndex} type with

In [52]:
\texttt{e.m} = [\texttt{CartesianIndex}((1:M .== i)*1\ldots) \texttt{for i in 1:M]}

Out[52]: 3-element Array\{CartesianIndex\{3\},1\}:
\texttt{CartesianIndex(1, 0, 0)}
\texttt{CartesianIndex(0, 1, 0)}
\texttt{CartesianIndex(0, 0, 1)}
and then use the vector to increment. For example, if the current count is \((1, 2, 2)\) and we want to add a count of \(1\) to the first index and remove a count of \(1\) from the third index, then

In [53]: `ind = CartesianIndex(1, 2, 2)`  # example counts coming from
   CartesianIndices
  @show ind + e_m[1]  # increment the first index
  @show ind - e_m[3];  # decrement the third index

   \[
   \text{ind} + e_m[1] = \text{CartesianIndex}(2, 2, 2)
   \]
   \[
   \text{ind} - e_m[3] = \text{CartesianIndex}(1, 2, 1)
   \]

This works, of course, because the \texttt{CartesianIndex} type is written to support efficient addition and subtraction. Finally, to implement the operator, we need to count the indices in the states where increment and decrement occurs.

In [54]: `@show ind
   @show count(ind.I .> 1)
   @show count(ind.I .< N);

   \[
   \text{ind} = \text{CartesianIndex}(1, 2, 2)
   \]
   \[
   \text{count(ind.I .> 1)} = 2
   \]
   \[
   \text{count(ind.I .< N)} = 1
   \]

With this, we are now able to write the \(Q\) operator on the \(f\) vector, which is enumerated by the Cartesian indices. First, collect the parameters in a named tuple generator

In [55]: `using Parameters, BenchmarkTools
   default_params = @with_kw (θ = 0.1, ζ = 0.05, ρ = 0.03, N = 10, M = 6,
   shape = Tuple(fill(N, M)),  # for reshaping)
   e_m = ([CartesianIndex((1:M == i)*1...) for i in 1:M]))

Out[55]: ##NamedTuple_kw#256 (generic function with 2 methods)

Next, implement the in-place matrix-free product

In [56]: `function Q_mul!(dv, v, p)
   @unpack θ, ζ, N, M, shape, e_m = p
   v = reshape(v, shape)  # now can access v, dv as M-dim arrays
   dv = reshape(dv, shape)
   @inbounds for ind in CartesianIndices(v)
   dv[ind] = 0.0
   for m in 1:M
   n_m = ind[m]
   if(n_m < N)
   dv[ind] += θ * v[ind + e_m[m]]
   `
23.8. KRYLOV METHODS FOR MARKOV-CHAIN DYNAMICS

```plaintext
if(n_m > 1)
    dv[ind] += ζ * v[ind - e_m[m]]
end

dv[ind] -= (θ * count(ind.I .< N) + ζ * count(ind.I .> 1)) * v[ind]
end

p = default_params()
v = zeros(p.shape)
dv = similar(v)
@btime Q_mul!(dv, v, p)
  55.717 ms (0 allocations: 0 bytes)
```

From the output of the benchmarking, note that the implementation of the left-multiplication takes less than 100 milliseconds, and allocates little or no memory, even though the Markov chain has a million possible states (i.e., $N^M = 10^6$).

23.8.2 Solving a Valuation Problem

As before, we could use this Markov chain to solve a Bellman equation. Assume that the firm discounts at rate $ρ > 0$ and gets a flow payoff of a different $z_m$ per customer of type $m$. For example, if the state of the firm is $(n_1,n_2,n_3) = (2,3,2)$, then it gets $[2 3 2]$ in flow profits.

Given this profit function, we can write the simple Bellman equation in our standard form of $ρv = r + Qv$, defining the appropriate payoff $r$. For example, if $z_m = m^2$, then

```plaintext
In[57]: function r_vec(p)
    z = (1:p.M).^2  # payoffs per type m
    r = [0.5 * dot(ind.I, z) for ind in CartesianIndices(p.shape)]
    return reshape(r, p.N^p.M)  # return as a vector
end
@show typeof(r_vec(p))
```

```
Out[57]: 250.25
```

Note that the returned $r$ is a vector, enumerated in the same order as the $n_m$ states.

Since the ordering of $r$ is consistent with that of $Q$, we can solve $(ρ - Q)v = r$ as a linear system.

Below, we create a linear operator and compare the algorithm for a few different iterative methods (GMRES, BiCGStab1, IDR(s), etc.) with a small problem of only 10,000 possible states.
In [58]:

```python
p = default_params(N=10, M=4)
Q = LinearMap((df, f) -> Q_mul!(df, f, p), p.N*p.M, ismutating=True)
A = p.ρ * I - Q
A_sparse = sparse(A)  # expensive: use only in tests
r = r_vec(p)
v_direct = A_sparse \ r
iv = zero(r)

@btime $A_sparse \ r  # direct
@show norm(gmres(A, r) - v_direct)
@btime gmres!(iv, A, r) setup = (iv = zero(r))

@show norm(bicgstabl(A, r) - v_direct)
@btime bicgstabl!(iv, A, r) setup = (iv = zero(r))

@show norm(idrs(A, r) - v_direct)
@btime idrs(A, r);
```

800.500 ms (75 allocations: 181.85 MiB)

Here, we see that even if the A matrix has been created, the direct sparse solver (which uses a sparse LU or QR) is at least an order of magnitude slower and allocates over an order of magnitude more memory. This is in addition to the allocation for the A_sparse matrix itself, which is not needed for iterative methods.

The different iterative methods have tradeoffs when it comes to accuracy, speed, convergence rate, memory requirements, and usefulness of preconditioning. Going much above $N = 10^4$, the direct methods quickly become infeasible.

Putting everything together to solving much larger systems with GMRES as our linear solvers

In [59]:

```python
function solve_bellman(p; iv = zeros(p.N*p.M))
    @unpack ρ, N, M = p
    Q = LinearMap((df, f) -> Q_mul!(df, f, p), N*M, ismutating=True)
    A = ρ * I - Q
    r = r_vec(p)
    sol = gmres!(iv, A, r, log=false)  # iterative solver, matrix-free
    return sol
end
```

```
p = default_params(N=10, M=6)
@btime solve_bellman($p);
```

1.684 s (270 allocations: 366.23 MiB)

This solves a value function with a Markov chain of a million states in a little over a second! This general approach seems to scale roughly linearly. For example, try $N = 10, M = 8$ to solve an equation with a Markov chain with 100 million possible states, which can be solved
in about 3-4 minutes. Above that order of magnitude, you may need to tinker with the linear solver parameters to ensure that you are not memory limited (e.g., change the restart parameter of GMRES).

### 23.8.3 Markov Chain Steady State and Dynamics

Recall that given an \(N\)-dimensional intensity matrix \(Q\) of a CTMC, the evolution of the pdf from an initial condition \(\psi(0)\) is the system of linear differential equations

\[
\dot{\psi}(t) = Q^T \psi(t)
\]

If \(Q\) is a matrix, we could just take its transpose to find the adjoint. However, with matrix-free methods, we need to implement the adjoint-vector product directly.

The logic for the adjoint is that for a given \(n = (n_1, \ldots, n_m, \ldots n_M)\), the \(Q^T\) product for that row has terms enter when

1. \(1 < n_m \leq N\), entering into the identical \(n\) except with one less customer in the \(m\) position
2. \(1 \leq n_m < N\), entering into the identical \(n\) except with one more customer in the \(m\) position

Implementing this logic, first in math and then in code,

\[
Q^T_{(n_1, \ldots, n_M)} \cdot \psi = \theta \sum_{m=1}^{M} (n_m > 1) \psi(n_1, \ldots, n_m - 1, \ldots, n_M)
+ \zeta \sum_{m=1}^{M} (n_m < N) \psi(n_1, \ldots, n_m + 1, \ldots, n_M)
- (\theta \text{Count}(n_m < N) + \zeta \text{Count}(n_m > 1)) \psi(n_1, \ldots, n_M)
\]

In [60]: function Q_T_mul!(dψ, ψ, p)
    @unpack θ, ζ, N, M, shape, e_m = p
    ψ = reshape(ψ, shape)
    dψ = reshape(dψ, shape)
    @inbounds for ind in CartesianIndices(ψ)
        dψ[ind] = 0.0
        for m in 1:M
            n_m = ind[m]
            if(n_m > 1)
                dψ[ind] += θ * ψ[ind - e_m[m]]
            end
            if(n_m < N)
                dψ[ind] += ζ * ψ[ind + e_m[m]]
            end
        end
        dψ[ind] -= (θ * count(ind.I .< N) + ζ * count(ind.I .> 1)) * ψ[ind]
    end
end
The \texttt{sparse} function for the operator is useful for testing that the function is correct, and is the adjoint of our \texttt{Q} operator.

```
In [61]:
p = default_params(N=5, M=4)  # sparse is too slow for the full matrix
Q = LinearMap((df, f) -> Q_mul!(df, f, p), p.N^p.M, ismutating = true)
@show norm(sparse(Q)' - sparse(Q_T));  # reminder: use sparse only for testing!
norm((sparse(Q))' - sparse(Q_T)) = 0.0
```

As discussed previously, the steady state can be found as the eigenvector associated with the zero eigenvalue (i.e., the one that solves $Q^T \psi = 0 \psi$). We could do this with a dense eigenvalue solution for relatively small matrices

```
In [62]:
eig_Q_T = eigen(Matrix(Q_T))
vec = real(eig_Q_T.vectors[:,end])
direct_ψ = vec ./ sum(vec)
@show eig_Q_T.values[end];
eig_Q_T.values[end] = -4.163336342344337e-16 + 0.0im
```

This approach relies on a full factorization of the underlying matrix, delivering the entire spectrum. For our purposes, this is not necessary.

Instead, we could use the \texttt{Arpack.jl} package to target the eigenvalue of smallest absolute value, which relies on an iterative method.

A final approach in this case is to notice that the $N \times N$ matrix is of rank $N - 1$ when the Markov chain is irreducible. The stationary solution is a vector in the 1-dimensional nullspace of the matrix.

Using Krylov methods to solve a linear system with the right-hand side all 0 values will converge to a point in the nullspace. That is, $\min_x \|Ax - 0\|_2$ solved iteratively from a non-zero initial condition will converge to a point in the nullspace.

We can use various Krylov methods for this trick (e.g., if the matrix is symmetric and positive definite, we could use Conjugate Gradient) but in our case we will use GMRES since we do not have any structure.

```
In [63]:
p = default_params(N=5, M=4)  # sparse is too slow for the full matrix
ψ = fill(1/(p.N^p.M), p.N^p.M)  # can't use 0 as initial guess
sol = gmres!(ψ, Q_T, zeros(p.N^p.M))  # i.e., solve Ax = 0 iteratively
ψ = ψ / sum(ψ)
@show norm(ψ - direct_ψ);
norm(ψ - direct_ψ) = 6.098250476301026e-11
```
The speed and memory differences between these methods can be orders of magnitude.

In [64]: p = default_params(N=4, M=4)  # Dense and sparse matrices are too slow]
   for the full
   dataset.
   Q_T = LinearMap((dψ, ψ) -> Q_T_mul!(dψ, ψ, p, p.N^p.M, ismutating = true)
   Q_T_dense = Matrix(Q_T)
   Q_T_sparse = sparse(Q_T)
   b = zeros(p.N^p.M)
   @btime eigen($Q_T_dense)
   @btime eigs($Q_T_sparse, nev=1, which=:SM, v0 = iv) setup = (iv = fill(1/
   (p.N^p.M),

   43.285 ms (270 allocations: 2.28 MiB)
   5.317 ms (341 allocations: 1.21 MiB)
   205.602 μs (359 allocations: 66.03 KiB)

The differences become even more stark as the matrix grows. With default_params(N=5,
M=5), the gmres solution is at least 3 orders of magnitude faster, and uses close to 3 orders
of magnitude less memory than the dense solver. In addition, the gmres solution is about an
order of magnitude faster than the iterative sparse eigenvalue solver.

The algorithm can solve for the steady state of $10^5$ states in a few seconds

In [65]: function stationary_ψ(p)
   Q_T = LinearMap((dψ, ψ) -> Q_T_mul!(dψ, ψ, p, p.N^p.M, ismutating = true)
   ψ = fill(1/(p.N^p.M), p.N^p.M) # can't use 0 as initial guess
   sol = gmres!(ψ, Q_T, zeros(p.N^p.M)) # i.e., solve $Ax = 0$ iteratively
   return ψ / sum(ψ)
   end
   p = default_params(N=10, M=5)
   @btime stationary_ψ(p);

   3.374 s (4880 allocations: 19.32 MiB)

As a final demonstration, consider calculating the full evolution of the $ψ(t)$ Markov chain. For
the constant $Q'$ matrix, the solution to this system of equations is $ψ(t) = \exp(Q')ψ(0)$

Matrix-free Krylov methods using a technique called exponential integration can solve this for
high-dimensional problems.

For this, we can set up a MatrixFreeOperator for our Q_T_mul! function (equivalent
to the LinearMap, but with some additional requirements for the ODE solver) and use the
LinearExponential time-stepping method.

In [66]: using OrdinaryDiffEq, DiffEqOperators

   function solve_transition_dynamics(p, t)
       @unpack N, M = p
\( \psi_0 = \begin{bmatrix} 1.0; \text{fill}(0.0, N^M - 1) \end{bmatrix} \)

\( O! = \text{MatrixFreeOperator}((d\psi, \psi, p, t) \to Q_T \cdot \text{mul!}(d\psi, \psi, p), (p, 0.0), \text{size}=(N^M,N^M), \text{opnorm}(p)>1.25) \)

# define the corresponding ODE problem
prob = ODEProblem(O!, \psi_0, \begin{bmatrix} 0.0; \text{end} \end{bmatrix}, p)
return solve(prob, LinearExponential(krylov=simple), tstops = t)
end

t = 0.0:5.0:100.0
p = default_params(N=10, M=6)
sol = solve_transition_dynamics(p, t)
v = solve_bellman(p)
plot(t, [dot(sol(tval), v) for tval in t], xlabel = "t", label = "E_t(v)"")

Out[66]:

The above plot (1) calculates the full dynamics of the Markov chain from the \( n_m = 1 \) for all \( m \) initial condition; (2) solves the dynamics of a system of a million ODEs; and (3) uses the calculation of the Bellman equation to find the expected valuation during that transition. The entire process takes less than 30 seconds.
Part IV

Dynamic Programming
Chapter 24

Shortest Paths

24.1 Contents

- Overview 24.2
- Outline of the Problem 24.3
- Finding Least-Cost Paths 24.4
- Solving for J 24.5
- Exercises 24.6
- Solutions 24.7

24.2 Overview

The shortest path problem is a classic problem in mathematics and computer science with applications in

- Economics (sequential decision making, analysis of social networks, etc.)
- Operations research and transportation
- Robotics and artificial intelligence
- Telecommunication network design and routing
- etc., etc.

Variations of the methods we discuss in this lecture are used millions of times every day, in applications such as

- Google Maps
- routing packets on the internet

For us, the shortest path problem also provides a nice introduction to the logic of dynamic programming.

Dynamic programming is an extremely powerful optimization technique that we apply in many lectures on this site.

24.3 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
- A, C, F, G at cost 8
24.4 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
\[
\min_{w \in F_v} \{c(v, w) + J(w)\} \tag{1}
\]

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

\[
J(v) = \min_{w \in F_v} \{c(v, w) + J(w)\} \tag{2}
\]

This is known as the Bellman equation, after the mathematician Richard Bellman.

### 24.5 Solving for \( J \)

The standard algorithm for finding \( J \) is to start with

\[
J_0(v) = M \text{ if } v \neq \text{ destination, else } J_0(v) = 0 \tag{3}
\]

where \( M \) is some large number.

Now we use the following algorithm

1. Set \( n = 0 \).
2. Set \( J_{n+1}(v) = \min_{w \in F_v} \{c(v, w) + J_n(w)\} \) for all \( v \).
3. If \( J_{n+1} \) and \( J_n \) are not equal then increment \( n \), go to 2.

In general, this sequence converges to \( J \)—the proof is omitted.

### 24.6 Exercises

#### 24.6.1 Exercise 1

Use the algorithm given above to find the optimal path (and its cost) for the following graph.
In [3]: graph = Dict(zip(6, 20.59), (46, 1247.25), [(45, 1561.45), (31, 166.8), (66, 54.18)], [(11, 42.43), (60.2)], (26, 133.65), [(7, 1.02), (5, 0.73), (75, 3706.67)], [(11, 34.54), (7, 3.33), (45, 1382.97)], [(10, 13.1), (9, 0.72), (31, 63.17)], [(10, 5.85), (9, 3.15)], (50, 478.14), [(12, 3.18), (11, 7.45), (69, 577.91)], [(20, 16.53), (13, 4.42), (70, 2454.28)], [(16, 25.16), (12, 1.87), (89, 5352.79)], [(20, 65.08), (18, 37.55), (94, 4961.32)], [(28, 179.04), (24, 34.32), (84, 3914.62)], [(40, 475.33), (38, 236.33), (60, 2135.95)], [(24, 38.65), (16, 2.7), (67, 1878.96)], [(18, 2.57), (17, 1.01), (91, 3597.11)], [(38, 278.71), (19, 3.49), (36, 392.92)], [(23, 26.45), (22, 24.78), (76, 783.29)], [(28, 55.84), (23, 16.23), (91, 3363.17)], [(28, 70.54), (20, 0.24), (26, 20.09)], [(33, 145.8)], [(24, 9.81), (98, 3523.33)], [(31, 27.06), (28, 36.65), (56, 626.04)], [(40, 124.22)], [(39, 136.32), (72, 1447.22)], [(33, 22.37), (26, 2.66), (52, 336.73)], [(28, 14.25)], [(26, 1.8), (66, 875.19)], [(35, 45.55), (32, 36.58), (70, 1343.63)], [(42, 122.0), (27, 0.81)], [(47, 135.78)], [(43, 246.24), (35, 48.1), (65, 480.55)], [(36, 15.52), (34, 21.79)], [(82, 2538.18)], [(33, 12.61), (32, 4.22), (64, 635.52)], [(35, 13.95), (33, 5.61)], [(98, 2616.03)], [(44, 125.88), (36, 20.44), (98, 3350.98)], [(35, 1.46), (34, 3.33)], [(97, 2613.92)], [(47, 111.54), (41, 3.23), (81, 1854.73)], [(48, 129.45), (42, 51.52)], [(73, 1075.38)], [(50, 78.81), (41, 2.09), (52, 17.57)], [(57, 260.46), (54, 101.88)], [(71, 1171.6)], [(46, 80.49), (38, 0.36), (75, 269.97)], [(42, 8.78), (40, 1.79)], [(93, 2767.85)], [(41, 1.34), (40, 0.95), (50, 39.88)], [(54, 53.46), (47, 28.57)], [(75, 548.68)], [(54, 162.24), (46, 0.28), (53, 18.23)], [(72, 437.49), (47, 10.88)], [(59, 141.86)], [(60, 116.23), (54, 95.06), (98, 2984.83)], [(47, 2.14), (46, 1.56)], [(91, 807.39)], [(49, 15.51), (47, 3.68), (58, 79.93)], [(67, 65.48), (57, 27.5)], [(52, 22.68)], [(61, 172.64), (56, 49.31), (50, 2.82)], [(60, 66.44), (59, 34.52)], [(99, 2564.12)], [(56, 10.89), (50, 0.51), (78, 53.79)], [(55, 20.1), (53, 1.38)], [(85, 251.76)], [(60, 73.79), (59, 23.67), (98, 2110.67)], [(66, 123.03), (64, 102.41)], [(94, 1471.8)], [(67, 88.35), (56, 4.33), (72, 22.85)], [(73, 238.61), (59, 24.3)], [(88, 967.59)], [(64, 60.8), (57, 2.13), (84, 86.09)], [(61, 11.06), (57, 0.02)], [(76, 197.03)], [(60, 7.01), (58, 0.46), (86, 701.09)], [(65, 34.32), (64, 29.85), (83, 556.7)], [(71, 485)]
CHAPTER 24. SHORTEST PATHS

Out[3]: Dict[Int64,Array[Tuple[Int64,Float64],1]] with 100 entries:
   68 => [(71, 1.66), (70, 3.35), (94, 232.1)]
   2 => [(45, 1561.45), (31, 166.8), (66, 54.18)]
   89 => [(99, 82.12), (97, 36.97)]
   11 => [(20, 65.08), (18, 37.55), (94, 4961.32)]
   39 => [(41, 1.34), (40, 0.95), (50, 39.88)]
   46 => [(67, 65.48), (57, 27.5), (52, 22.68)]
   85 => [(89, 4.9), (87, 2.66), (97, 94.51)]
   25 => [(35, 45.55), (32, 36.58), (70, 1343.63)]
   55 => [(64, 60.8), (57, 2.13), (84, 86.09)]
   42 => [(72, 437.49), (47, 10.08), (59, 141.86)]
   29 => [(33, 12.61), (32, 4.22), (64, 635.52)]
   58 => [(65, 34.32), (64, 29.85), (83, 556.7)]
   66 => [(73, 37.53), (68, 2.66), (98, 395.63)]
   59 => [(71, 0.67), (60, 0.72), (90, 828.66)]
   8 => [(12, 3.18), (11, 7.45), (69, 577.91)]
   74 => [(78, 8.08), (76, 0.52), (81, 1.67)]
   95 => [(99, 0.27), (98, 6.17)]
   57 => [(60, 7.01), (58, 0.46), (86, 701.09)]
   20 => [(33, 145.8), (24, 9.81), (98, 3523.33)]
   90 => [(99, 50.99), (94, 10.47), (96, 23.53)]
   14 => [(24, 38.65), (16, 2.7), (67, 1878.96)]
   31 => [(44, 125.88), (36, 20.44), (98, 3350.98)]
   78 => [(81, 2.59), (98, 355.9)]
The cost from node 68 to node 71 is 1.66 and so on.

### 24.7 Solutions

#### 24.7.1 Exercise 1

In [4]:

```jupyter
function update_J!(J, graph)
    next_J = Dict()
    for node in keys(graph)
        if node == 99
            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 = 0.0
    current_location, destination = extrema(keys(graph))
    while current_location != destination
        println("node $current_location")
        running_min = 1e10
        minimizer_dest = Inf
        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
    sum_costs = round(sum_costs, digits = 2)
    println("node $destination\nCost: $sum_costs")
end

J = Dict((node => Inf) for node in keys(graph))
while true
    next_J = update_J!(J, graph)
    if next_J == J
        break
    end
```

```
```python
else
    J = next_J
end

print_best_path(J, graph)

node 0
node 8
node 11
node 18
node 23
node 33
node 41
node 53
node 56
node 57
node 60
node 67
node 70
node 73
node 76
node 85
node 87
node 88
node 93
node 94
node 96
node 97
node 98
node 99
Cost: 160.55
```
Chapter 25

Job Search I: The McCall Search Model

25.1 Contents

- Overview 25.2
- The McCall Model 25.3
- Computing the Optimal Policy: Take 1 25.4
- Computing the Optimal Policy: Take 2 25.5
- Exercises 25.6
- Solutions 25.7

“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.

25.2 Overview

The McCall search model [76] 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 McCall’s model and adopt the same solution method.

As we’ll see, McCall’s model is not only interesting in its own right but also an excellent vehicle for learning dynamic programming.
25.3 The McCall Model

An unemployed worker receives in each period a job offer at wage $W_t$. At time $t$, our worker has two choices:

1. Accept the offer and work permanently at constant wage $W_t$.
2. Reject the offer, receive unemployment compensation $c$, and reconsider next period.

The wage sequence $\{W_t\}$ is assumed to be iid with probability mass function $p_1, \ldots, p_n$. Here $p_i$ is the probability of observing wage offer $W_t = w_i$ in the set $w_1, \ldots, w_n$.

The worker is infinitely lived and aims to maximize the expected discounted sum of earnings:

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t Y_t$$

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

25.3.1 A Trade Off

The worker faces a trade-off:

- Waiting too long for a good offer is costly, since the future is discounted.
- Accepting too early is costly, since better offers might arrive in the future.

To decide optimally in the face of this trade off, we use dynamic programming. Dynamic programming can be thought of as a two step procedure that

1. first assigns values to “states” and
2. then deduces optimal actions given those values

We’ll go through these steps in turn.

25.3.2 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 (1) when an agent in this situation makes optimal decisions now and at all future points in time.

Of course $V(w)$ is not trivial to calculate because we don’t yet know what decisions are optimal and what aren’t!

But think of $V$ as a function that assigns to each possible wage $w$ the maximal lifetime value that can be obtained with that offer in hand.

A crucial observation is that this function $V$ must satisfy the recursion

$$V(w) = \max \left\{ \frac{w}{1 - \beta}, c + \beta \sum_{i=1}^{n} V(w_i)p_i \right\}$$

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 (1).

### 25.3.3 The Optimal Policy

Suppose for now that we are able to solve (1) 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 (1).

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 (1).

Thus, we have a map from $\mathbb{R}$ to $\{0, 1\}$, with 1 meaning accept and zero meaning reject.
We can write the policy as follows

\[ \sigma(w) := 1 \left\{ \frac{w}{1-\beta} \geq c + \beta \sum_{i=1}^{n} V(w_i)p_i \right\} \]

Here \( 1\{P\} = 1 \) if statement \( P \) is true and equals zero otherwise.

We can also write this as

\[ \sigma(w) := 1\{w \geq \bar{w}\} \]

where

\[ \bar{w} := (1-\beta) \left\{ c + \beta \sum_{i=1}^{n} V(w_i)p_i \right\} \] (2)

Here \( \bar{w} \) is a constant depending on \( \beta, c \) and the wage distribution, called the reservation wage.

The agent should accept if and only if the current wage offer exceeds the reservation wage.

Clearly, we can compute this reservation wage if we can compute the value function.

### 25.4 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 (1), this vector satisfies the nonlinear system of equations

\[ v_i = \max \left\{ \frac{w_i}{1-\beta}, c + \beta \sum_{i=1}^{n} v_ip_i \right\} \text{ for } i = 1, \ldots, n \] (3)

It turns out that there is exactly one vector \( v := (v_i)_{i=1}^{n} \) in \( \mathbb{R}^n \) that satisfies this equation.

#### 25.4.1 The Algorithm

To compute this vector, we proceed as follows:

Step 1: pick an arbitrary initial guess \( v \in \mathbb{R}^n \).

Step 2: compute a new vector \( v' \in \mathbb{R}^n \) via

\[ v'_i = \max \left\{ \frac{w_i}{1-\beta}, c + \beta \sum_{i=1}^{n} v_ip_i \right\} \text{ for } i = 1, \ldots, n \] (4)

Step 3: calculate a measure of the deviation between \( v \) and \( v' \), such as \( \max_i |v_i - v'_i| \).

Step 4: if the deviation is larger than some fixed tolerance, set \( v = v' \) and go to step 2, else continue.
Step 5: return $v$.

This algorithm returns an arbitrarily good approximation to the true solution to (3), which represents the value function.

(Arbitrarily good means here that the approximation converges to the true solution as the tolerance goes to zero)

### 25.4.2 The Fixed Point Theory

What’s the math behind these ideas?

First, one defines a mapping $T$ from $\mathbb{R}^n$ to itself via

$$
Tv_i = \max \left\{ \frac{w_i}{1 - \beta}, c + \beta \sum_{i=1}^{n} v_i p_i \right\} \quad \text{for } i = 1, \ldots, n
$$

(5)

(A new vector $Tv$ is obtained from given vector $v$ by evaluating the r.h.s. at each $i$)

One can show that the conditions of the Banach contraction mapping theorem are satisfied by $T$ as a self-mapping on $\mathbb{R}^n$.

One implication is that $T$ has a unique fixed point in $\mathbb{R}^n$.

Moreover, it’s 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.

### 25.4.3 Implementation

#### 25.4.4 Setup

In [1]:

```julia
using InstantiateFromURL
# optionally add arguments to force installation: instantiate = true,
~
precompile = true

github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")
```

In [2]:

```julia
using LinearAlgebra, Statistics
using Distributions, Expectations, NLsolve, Roots, Random, Plots, Parameters
```

In [3]:

```julia
gr(fmt = :png);
```

Here’s the distribution of wage offers we’ll work with

In [4]:

```julia
n = 50

dist = BetaBinomial(n, 200, 100) # probability distribution
@show support(dist)

w = range(10.0, 60.0, length = n+1) # linearly space wages
```
We can explore taking expectations over this distribution

In [5]: E = expectation(dist)  # expectation operator

# exploring the properties of the operator
wage(i) = w[i+1]  # +1 to map from support of 0
E_w = E(wage)
E_w_2 = E(i -> wage(i)^2) - E_w^2  # variance
@show E_w, E_w_2

# use operator with left-multiply
@show E * w  # the `w` are values assigned for the discrete states
@show dot(pdf.(dist, support(dist)), w);  # identical calculation

(E_w, E_w_2) = (43.33333333335695, 12.919896640724573)
E * w = 43.333333333357
dot(pdf.(dist, support(dist)), w) = 43.333333333357

To implement our algorithm, let’s have a look at the sequence of approximate value functions that this fixed point algorithm generates.
Default parameter values are embedded in the function.

Our initial guess $v$ is the value of accepting at every given wage

In [6]: # parameters and constant objects

c = 25
β = 0.99
num_plots = 6

# Operator
$T(v) = \max\left(w/(1 - \beta), c + \beta \cdot E\cdot v\right)$ # (5) broadcasts over the w, fixes the v
# alternatively, $T(v) = \left[\max(wval/(1 - \beta), c + \beta \cdot E\cdot v)\right]$ for wval in w

# fill in matrix of vs
vs = zeros(n + 1, 6) # data to fill
vs[:, 1] = w / (1 - β) # initial guess of "accept all"

# manually applying operator
for col in 2:num_plots
    v_last = vs[:, col - 1]
    vs[:, col] = T(v_last) # apply operator
end
plot(vs)

Out[6]:

One approach to solving the model is to directly implement this sort of iteration, and continue until measured deviation between successive iterates is below tol

In [7]: function compute_reservation_wage_direct(params; v_iv = collect(w ./ (1 - β)), max_iter =

500,
CHAPTER 25. JOB SEARCH I: THE MCCALL SEARCH MODEL

tol = 1e-6)

@unpack c, β, w = params

# create a closure for the T operator
T(v) = max.(w/(1 - β), c + β * E*v) # (5) fixing the parameter values

v = copy(v_iv) # start at initial value. copy to prevent v_iv modification
v_next = similar(v)
i = 0
error = Inf
while i < max_iter && error > tol
    v_next .= T(v) # (4)
    error = norm(v_next - v)
    i += 1
    v .= v_next # copy contents into v. Also could have used v[:] =
end

# now compute the reservation wage
return (1 - β) * (c + β * E*v) # (2)
end

Out[7]: compute_reservation_wage_direct (generic function with 1 method)

In the above, we use v = copy(v_iv) rather than just v_iv = v.

To understand why, first recall that v_iv is a function argument – either defaulting to the
given value, or passed into the function

- If we had gone v = v_iv instead, then it would have simply created a new
  name v which binds to whatever is located at v_iv.
- Since we later use v .= v_next later in the algorithm, the values in it would be modi-
  fied.
- Hence, we would be modifying the v_iv vector we were passed in, which may not be
  what the caller of the function wanted.
- The big issue this creates are “side-effects” where you can call a function and strange
  things can happen outside of the function that you didn’t expect.
- If you intended for the modification to potentially occur, then the Julia style guide says
  that we should call the function compute_reservation_wage_direct! to make
  the possible side-effects clear.

As usual, we are better off using a package, which may give a better algorithm and is likely to
less error prone.

In this case, we can use the fixedpoint algorithm discussed in our Julia by Example lec-
ture to find the fixed point of the T operator.

In [8]: function compute_reservation_wage(params; v_iv = collect(w ./((1-β))),
   iterations = 500,
   ftol = 1e-6, m = 6)
   @unpack c, β, w = params
   T(v) = max.(w/(1 - β), c + β * E*v) # (5) fixing the parameter values
   v_star = fixedpoint(T, v_iv, iterations = iterations, ftol = ftol,
25.4. COMPUTING THE OPTIMAL POLICY: TAKE 1

```python
m = 0).zero  # (5)
return (1 - β) * (c + β * E^v_star)  # (3)
end
```

Out[8]: `compute_reservation_wage (generic function with 1 method)`

Let’s compute the reservation wage at the default parameters

In [9]: `mcm = @with_kw (c=25.0, β=0.99, w=w) # named tuples
compute_reservation_wage(mcm()) # call with default parameters`

Out[9]: `47.31649970147162`

25.4.5 Comparative Statics

Now we know how to compute the reservation wage, let’s see how it varies with parameters. In particular, let’s look at what happens when we change $\beta$ and $c$.

In [10]: `grid_size = 25
R = rand(grid_size, grid_size)
c_vals = range(10.0, 30.0, length = grid_size)
β_vals = range(0.9, 0.99, length = grid_size)

for (i, c) in enumerate(c_vals)
    for (j, β) in enumerate(β_vals)
        R[i, j] = compute_reservation_wage(mcm(c=c, β=β)) # change from defaults
end
end`

In [11]: `contour(c_vals, β_vals, R',
title = "Reservation Wage",
xlabel = "c",
ylabel = "beta",
fill = true)`

Out[11]:
As expected, the reservation wage increases both with patience and with unemployment compensation.

25.5 Computing the Optimal Policy: Take 2

The approach to dynamic programming just described is very standard and broadly applicable.

For this particular problem, there’s also an easier way, which circumvents the need to compute the value function.

Let $\psi$ denote the value of not accepting a job in this period but then behaving optimally in all subsequent periods.

That is,

$$\psi = c + \beta \sum_{i=1}^{n} V(w_i)p_i$$  \hspace{1cm} (6)

where $V$ is the value function.

By the Bellman equation, we then have

$$V(w_i) = \max \left\{ \frac{w_i}{1-\beta}, \psi \right\}$$

Substituting this last equation into (6) gives

$$\psi = c + \beta \sum_{i=1}^{n} \max \left\{ \frac{w_i}{1-\beta}, \psi \right\} p_i$$  \hspace{1cm} (7)
Which we could also write as $ψ = T_ψ(ψ)$ for the appropriate operator.

This is a nonlinear equation that we can solve for $ψ$.

One solution method for this kind of nonlinear equation is iterative.

That is,

Step 1: pick an initial guess $ψ$.
Step 2: compute the update $ψ'$ via

$$ψ' = c + β \sum_{i=1}^{n} \max \left\{ \frac{w_i}{1 - β}, ψ \right\} p_i$$  \hspace{1cm} (8)

Step 3: calculate the deviation $|ψ − ψ'|$.
Step 4: if the deviation is larger than some fixed tolerance, set $ψ = ψ'$ and go to step 2, else continue.
Step 5: return $ψ$.

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’re iterating on a single number, rather than an $n$-vector.

Here’s an implementation:

```python
In [12]: function compute_reservation_wage_ψ(c, β; ψ_iv = E * w ./ (1 - β), ¶
       max_iter = 500, tol = 1e-5)
   T_ψ(ψ) = [c + β * E*max.((w ./ (1 - β)), ψ[1])] # (7)
   # using vectors since fixedpoint doesn’t support scalar
   ψ_star = fixedpoint(T_ψ, [ψ_iv]).zero[1]
   return (1 - β) * ψ_star # (2)
end
```

```python
Out[12]: 47.316499766546194
```

You can use this code to solve the exercise below.

Another option is to solve for the root of the $T_ψ(ψ) − ψ$ equation

```python
In [13]: function compute_reservation_wage_ψ_2(c, β; ψ_iv = E * w ./ (1 - β), ¶
       max_iter = 500, tol = 1e-5)
   root_ψ(ψ) = c + β * E*max.((w ./ (1 - β)), ψ) - ψ # (7)
   ψ_star = find_zero(root_ψ, ψ_iv)
   return (1 - β) * ψ_star # (2)
end
```

```python
Out[13]: 47.316499766546194
```
25.6 Exercises

25.6.1 Exercise 1

Compute the average duration of unemployment when $\beta = 0.99$ and $c$ takes the following values

$$c\_vals = \text{range}(10, 40, \text{length} = 25)$$

That is, start the agent off as unemployed, computed their reservation wage given the parameters, and then simulate to see how long it takes to accept.

Repeat a large number of times and take the average.

Plot mean unemployment duration as a function of $c$ in $c\_vals$.

25.7 Solutions

25.7.1 Exercise 1

Here’s one solution

```
In [14]: function compute_stopping_time(\bar{w}; seed=1234)
    Random.seed!(seed)
    stopping_time = 0
    t = 1
    # make sure the constraint is sometimes binding
    @assert length(w) - 1 ≥ support(dist) && \bar{w} <= w[end]
    while true
        # Generate a wage draw
        w_val = w[rand(dist)] # the wage dist set up earlier
        if w_val ≥ \bar{w}
            stopping_time = t
            break
        else
            t += 1
        end
    end
    return stopping_time
end

compute_mean_stopping_time(\bar{w}, num_reps=10000) = mean(i ->
    compute_stopping_time(\bar{w},
    seed = i), 1:num_reps)

c\_vals = range(10, 40, length = 25)
stop_times = similar(c\_vals)
beta = 0.99
for (i, c) in enumerate(c\_vals)
    \bar{w} = compute_reservation_wage_\psi(c, beta)
    stop_times[i] = compute_mean_stopping_time(\bar{w})
```
plot(c_vals, stop_times, label = "mean unemployment duration",
xlabel = "unemployment compensation", ylabel = "months")

Out[14]:

![Graph showing the relationship between unemployment compensation and mean unemployment duration. The graph has a blue line with steps indicating the mean duration across different compensation levels. The x-axis represents unemployment compensation, ranging from 10 to 40, and the y-axis represents months, ranging from 10 to 25.](image-url)
Chapter 26

Job Search II: Search and Separation

26.1 Contents

- Overview 26.2
- The Model 26.3
- Solving the Model using Dynamic Programming 26.4
- Implementation 26.5
- The Reservation Wage 26.6
- Exercises 26.7
- Solutions 26.8

26.2 Overview

Previously we looked at the McCall job search model [76] 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

26.2.1 Setup

In [1]: using InstantiateFromURL
    # optionally add arguments to force installation: instantiate = true, ...
    precompile = true
    github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

In [2]: using LinearAlgebra, Statistics
    using Distributions, Expectations, Parameters, NLsolve, Plots
26.3 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

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(Y_t)$$  \hspace{1cm} (1)

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 $u'' < 0$.

26.3.1 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)

26.4 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 (1) 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

$$V(w) = u(w) + \beta[(1 - \alpha)V(w) + \alpha U]$$

(2)

and

$$U = u(c) + \beta \sum_i \max\{U, V(w_i)\} p_i$$

(3)

Let’s interpret these two equations in light of the fact that today’s tomorrow is tomorrow’s today.

- The left hand sides of equations (2) and (3) 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) 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 (2) and (3) are the Bellman equations for this model.

Equations (2) and (3) provide enough information to solve out for both $V$ and $U$.

Before discussing this, however, let’s make a small extension to the model.

### 26.4.1 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

$$V(w) = u(w) + \beta[(1 - \alpha)V(w) + \alpha U]$$

(4)

and

$$U = u(c) + \beta(1 - \gamma)U + \beta\gamma \sum_i \max\{U, V(w_i)\} p_i$$

(5)
26.4.2 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 (4) and (5)
3. update the left hand sides from this rule and then repeat

In other words, we are iterating using the rules

$$V_{n+1}(w_i) = u(w_i) + \beta[(1 - \alpha)V_n(w_i) + \alpha U_n] \tag{6}$$

and

$$U_{n+1} = u(c) + \beta(1 - \gamma)U_n + \beta\gamma\sum_i \max\{U_n, V_n(w_i)\}p_i \tag{7}$$

starting from some initial conditions $U_0, V_0$.

Formally, we can define a “Bellman operator” $T$ which maps:

$$TV(\cdot) = u(\cdot) + \beta(1 - \alpha)V(\cdot) + \alpha U \tag{8}$$

In which case we are searching for a fixed point

$$TV^* = V^* \tag{9}$$

As before, the system always converges to the true solutions—in this case, the $V$ and $U$ that solve (4) and (5).

A proof can be obtained via the Banach contraction mapping theorem.

26.5 Implementation

Let’s implement this iterative process

```plaintext
In [3]: using Distributions, LinearAlgebra, Expectations, Parameters, NLsolve, Plots

function solve_mccall_model(mcm; U_iv = 1.0, V_iv = ones(length(mcm.w)),
                            tol = 1e-5,
                            iter = 2_000)
    # α, β, σ, c, γ, w = mcm.α, mcm.β, mcm.σ, mcm.c, mcm.γ, mcm.w
    @unpack α, β, σ, c, γ, w, dist, u = mcm
    # parameter validation
```
@assert c > 0.0
@assert minimum(w) > 0.0 # perhaps not strictly necessary, but useful

Here

# necessary objects
u_w = u(w, σ)
u_c = u(c, σ)
E = expectation(dist) # expectation operator for wage distribution

# Bellman operator T. Fixed point is x* s.t. T(x*) = x*
function T(x)
    V = x[1:end-1]
    U = x[end]
    [u_w + β * ((1 - α) * V + α * U); u_c + β * (1 - γ) * U + β * γ * E]
end

# value function iteration
x_iv = [V_iv; U_iv] # initial x val
xstar = fixedpoint(T, x_iv, iterations = iter, xtol = tol, m = 0).zero
V = xstar[1:end-1]
U = xstar[end]

# compute the reservation wage
wbarindex = searchsortedfirst(V - U, 0.0)
if wbarindex >= length(w) # if this is true, you never want to accept
    w = Inf
else
    w = w[wbarindex] # otherwise, return the number
end

# return a NamedTuple, so we can select values by name
return (V = V, U = U, w = w)
end

Out[3]: solve_mccall_model (generic function with 1 method)

The approach is to iterate until successive iterates are closer together than some small tolerance level.

We then return the current iterate as an approximate solution.

Let’s plot the approximate solutions U and V to see what they look like.

We’ll use the default parameterizations found in the code above.

In [4]: # a default utility function
u(c, σ) = (c^(1 - σ) - 1) / (1 - σ)

# model constructor
MccallModel = @with_kw (α = 0.2,
    β = 0.98, # discount rate
    γ = 0.7,
    c = 6.0, # unemployment compensation
    σ = 2.0,
\( u = u, \quad \# \text{utility function} \)

\( w = \text{range}(10, 20, \text{length} = 60), \quad \# \text{wage values} \)

\( \text{dist} = \text{BetaBinomial}(59, 600, 400) \quad \# \text{distribution over wage values} \)

Out[4]: ##NamedTuple_kw#254 (generic function with 2 methods)

In [5]: # plots setting
   \[
   \text{gr(fmt=:png);}
   \]

   \[
   \text{mcm} = \text{McCallModel();}
   \]

   \[
   @\text{unpack} \quad V, \quad U = \text{solve_mccall_model(mcm)}
   \]

   \[
   \text{U_vec} = \text{fill}(U, \text{length(mcm.w))}
   \]

   \[
   \text{plot(mcm.w, [V U_vec], lw = 2, \alpha = 0.7, \text{label} = ["V" "U"]}
   \]

Out[5]:

The value \( V \) is increasing because higher \( w \) generates a higher wage flow conditional on staying employed.

At this point, it’s natural to ask how the model would respond if we perturbed the parameters.

These calculations, called comparative statics, are performed in the next section.

26.6 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

If \( V(w) < U \) for all \( w \), then the function returns \( \text{np.inf} \).

Let’s use it to look at how the reservation wage varies with parameters.

In each instance below we’ll show you a figure and then ask you to reproduce it in the exercises.

### 26.6.1 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 tuple, 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.

### 26.6.2 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 $\beta$.

![Graph showing the reservation wage as a function of $\beta$.](image)

Again, the results are intuitive: More patient workers will hold out for higher wages.

### 26.6.3 The Reservation Wage and Job Destruction

Finally, let’s look at how $\bar{w}$ varies with the job separation rate $\alpha$.

Higher $\alpha$ translates to a greater chance that a worker will face termination in each period once employed.

![Graph showing the reservation wage as a function of $\alpha$.](image)

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.
26.7 Exercises

26.7.1 Exercise 1

Reproduce all the reservation wage figures shown above.

26.7.2 Exercise 2

Plot the reservation wage against the job offer rate $\gamma$.

Use

In [6]: $\gamma_{vals} = \text{range}(0.05, 0.95, \text{length} = 25)$

Out[6]: 0.05:0.0375:0.95

Interpret your results.

26.8 Solutions

26.8.1 Exercise 1

Using the solve_mccall_model 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

In [7]: $c_{vals} = \text{range}(2, 12, \text{length} = 25)$

models = [McCallModel(c = cval) for cval in c_vals]
sols = solve_mccall_model(models)
$\bar{w}_{vals} = [\text{sol.$\bar{w}$ for sol in sols}]

plot(c_vals,
     $\bar{w}_{vals},
     lw = 2,
     $\alpha = 0.7,
     xlabel = "unemployment compensation",
     ylabel = "reservation wage",
     label = "$\bar{w}$ as a function of $c$"
)

Out[7]:
Note that we could’ve done the above in one pass (which would be important if, for example, the parameter space was quite large).

In [8]: `w_vals = [solve_mccall_model(McCallModel(c = cval)).w for cval in c_vals];`  
   
   # doesn't allocate new arrays for models and solutions

26.8.2 Exercise 2

Similar to above, we can plot $\bar{w}$ against $\gamma$ as follows

In [9]: `γ_vals = range(0.05, 0.95, length = 25)`

   `models = [McCallModel(γ = γval) for γval in γ_vals]`
   `sols = solve_mccall_model(models)`
   `w_vals = [sol.w for sol in sols]`

   `plot(γ_vals, w_vals, lw = 2, α = 0.7, xlabel = "job offer rate",`  
   `ylabel = "reservation wage", label = "$\bar{w}$ 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.
Chapter 27

A Problem that Stumped Milton Friedman

(and that Abraham Wald solved by inventing sequential analysis)

27.1 Contents

• Overview 27.2
• Origin of the problem 27.3
• A dynamic programming approach 27.4
• Implementation 27.5
• Comparison with Neyman-Pearson formulation 27.6

Co-authored with Chase Coleman

27.2 Overview

This lecture describes a statistical decision problem encountered by Milton Friedman and W. Allen Wallis during World War II when they were analysts at the U.S. Government’s Statistical Research Group at Columbia University.

This problem led Abraham Wald [106] 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
27.3 Origin of the problem

On pages 137-139 of his 1998 book *Two Lucky People* with Rose Friedman [33], Milton Friedman described a problem presented to him and Allen Wallis during World War II, when they worked at the US Government’s Statistical Research Group at Columbia University.

Let’s listen to Milton Friedman tell us what happened.

“In order to understand the story, it is necessary to have an idea of a simple statistical problem, and of the standard procedure for dealing with it. The actual problem out of which sequential analysis grew will serve. The Navy has two alternative designs (say A and B) for a projectile. It wants to determine which is superior. To do so it undertakes a series of paired firings. On each round it assigns the value 1 or 0 to A accordingly as its performance is superior or 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 [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* [106].

We’ll formulate the problem using dynamic programming.

27.4 A dynamic programming approach

The following presentation of the problem closely follows Dmitri Berskekas’s treatment in *Dynamic Programming and Stochastic Control* [11].

A decision maker observes iid draws of a random variable $z$.

He (or she) wants to know which of two probability distributions $f_0$ or $f_1$ governs $z$.

After a number of draws, also to be determined, he makes a decision as to which of the distributions is generating the draws he observers.

To help formalize the problem, let $x \in \{x_0, x_1\}$ be a hidden state that indexes the two distributions:

$$
\mathbb{P}\{z = v \mid x\} =
\begin{cases}
  f_0(v) & \text{if } x = x_0, \\
  f_1(v) & \text{if } x = x_1
\end{cases}
$$

Before observing any outcomes, the decision maker believes that the probability that $x = x_0$ is
\[ p_{-1} = \mathbb{P}\{x = x_0 \mid \text{no observations}\} \in (0, 1) \]

After observing \( k + 1 \) observations \( z_k, z_{k-1}, \ldots, z_0 \), he updates this value to

\[ p_k = \mathbb{P}\{x = x_0 \mid z_k, z_{k-1}, \ldots, z_0\}, \]

which is calculated recursively by applying Bayes' law:

\[ p_{k+1} = \frac{p_k f_0(z_{k+1})}{p_k f_0(z_{k+1}) + (1 - p_k) f_1(z_{k+1})}, \quad k = -1, 0, 1, \ldots \]

After observing \( z_k, z_{k-1}, \ldots, z_0 \), the decision maker believes that \( z_{k+1} \) has probability distribution

\[ f(v) = p_k f_0(v) + (1 - p_k) f_1(v) \]

This is a mixture of distributions \( f_0 \) and \( f_1 \), with the weight on \( f_0 \) being the posterior probability that \( x = x_0 \) Section ??.

To help illustrate this kind of distribution, let's inspect some mixtures of beta distributions.

The density of a beta probability distribution with parameters \( a \) and \( b \) is

\[ f(z; a, b) = \frac{\Gamma(a + b) z^{a-1}(1 - z)^{b-1}}{\Gamma(a)\Gamma(b)} \quad \text{where} \quad \Gamma(t) := \int_0^\infty x^{t-1}e^{-x}dx \]

We'll discretize this distribution to make it more straightforward to work with.

The next figure shows two discretized beta distributions in the top panel.

The bottom panel presents mixtures of these distributions, with various mixing probabilities \( p_k \).

### 27.4.1 Setup

**In [1]:** using InstantiateFromURL

# optionally add arguments to force installation: instantiate = true,

| precompile = true |
| github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0") |

**In [2]:** using LinearAlgebra, Statistics

using Distributions, Parameters, Printf, Random, Roots, Plots

gf(fmt = :png)

**In [3]:** using StatsPlots

begin

base_dist = [Beta(1, 1), Beta(3, 3)]

mixed_dist = MixtureModel(Ref(base_dist), (p -> [p, one(p) - p]).(0.25:0.25:0.75))

plot(plot(base_dist, labels = ["f_0", "f_1"], title = "Original Distributions"),
27.4.2 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 \( z \).

27.4.3 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
27.4. A DYNAMIC PROGRAMMING APPROACH

- 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.

27.4.4 Intuition

Let’s try to guess what an optimal decision rule might look like before we go further.

Suppose at some given point in time that $p$ is close to 1.

Then our prior beliefs and the evidence so far point strongly to $x = x_0$.

If, on the other hand, $p$ is close to 0, then $x = x_1$ is strongly favored.

Finally, if $p$ is in the middle of the interval $[0, 1]$, then we have little information in either direction.

This reasoning suggests a decision rule such as the one shown in the figure

As we’ll see, this is indeed the correct form of the decision rule.

The key problem is to determine the threshold values $\alpha, \beta$, 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 $\beta$.

27.4.5 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

$$J(p) = \min \{(1 - p)L_0, \ pL_1, \ c + E[J(p')]\}$$

(1)

where $p'$ is the random variable defined by

$$p' = \frac{pf_0(z)}{pf_0(z) + (1 - p)f_1(z)}$$

(2)

when $p$ is fixed and $z$ is drawn from the current best guess, which is the distribution $f$ defined by

$$f(v) = pf_0(v) + (1 - p)f_1(v)$$

(3)

In the Bellman equation, minimization is over three actions:
1. accept $x_0$
2. accept $x_1$
3. postpone deciding and draw again

Let

$$A(p) := \mathbb{E}[J(p')]$$

Then we can represent the Bellman equation as

$$J(p) = \min\{(1 - p)L_0, pL_1, c + A(p)\}$$

where $p \in [0, 1]$.

Here

- $(1 - p)L_0$ is the expected loss associated with accepting $x_0$ (i.e., the cost of making a type II error).
- $pL_1$ is the expected loss associated with accepting $x_1$ (i.e., the cost of making a type I error).
- $c + A(p)$ is the expected cost associated with drawing one more $z$.

The optimal decision rule is characterized by two numbers $\alpha, \beta \in (0, 1) \times (0, 1)$ that satisfy

$$(1 - p)L_0 < \min\{pL_1, c + A(p)\} \text{ if } p \geq \alpha$$

and

$$pL_1 < \min\{(1 - p)L_0, c + A(p)\} \text{ if } p \leq \beta$$

The optimal decision rule is then

- accept $x = x_0$ if $p \geq \alpha$
- accept $x = x_1$ if $p \leq \beta$
- draw another $z$ if $\beta \leq p \leq \alpha$

Our aim is to compute the value function $J$, and from it the associated cutoffs $\alpha$ and $\beta$.

One sensible approach is to write the three components of $J$ that appear on the right side of the Bellman equation as separate functions.

Later, doing this will help us obey the don’t repeat yourself (DRY) golden rule of coding.

### 27.5 Implementation

Let’s code this problem up and solve it.

We implement the cost functions for each choice considered in the Bellman equation (7).
First, consider the cost associated to accepting either distribution and compare the minimum of the two to the expected benefit of drawing again.

Drawing again will only be worthwhile if the expected marginal benefit of learning from an additional draw is greater than the explicit cost.

For every belief \( p \), we can compute the difference between accepting a distribution and choosing to draw again.

The solution \( \alpha, \beta \) occurs at indifference points.

Define the cost function be the minimum of the pairwise differences in cost among the choices.

Then we can find the indifference points when the cost function is zero.

We can use any roots finding algorithm to solve for the solutions in the interval \([0, 1]\).

Lastly, verify which indifference points correspond to the definition of a permanent transition between the accept and reject space for each choice.

Here’s the code

```plaintext
In [4]:

accept_x0(p, L0) = (one(p) - p) * L0
accept_x1(p, L1) = p * L1
bayes_update(p, d0, d1) = p * pdf(d0, p) / pdf(MixtureModel([d0, d1], [p, one(p) - p]), p)
function draw_again(p, d0, d1, L0, L1, c, target)
    candidate = 0.0
    cost = 0.0
    while candidate < target
        p = bayes_update(p, d0, d1)
        cost += c
        candidate = min(accept_x0(p, L0), accept_x1(p, L1)) + cost
        if candidate >= target
            break
        end
    end
    target = candidate
end
return candidate
end
function choice(p, d0, d1, L0, L1, c)
    if isone(p)
        output = (1, 0)
    elseif iszero(p)
        output = (2, 0)
    elseif zero(p) < p < one(p)
        target, option = findmin([accept_x0(p, L0), accept_x1(p, L1)])
        candidate = draw_again(p, d0, d1, L0, L1, c, target)
        if candidate < target
            target, option = (candidate, 3)
        end
        output = (option, target)
    else
        throw(ArgumentError("p must be \# [0, 1]"))
    end
end
return output
end
```
Out[4]: choice (generic function with 1 method)

Next we solve a problem by finding the $\alpha$, $\beta$ values for the decision rule

In [5]: function decision_rule(d0, d1, L0, L1, c)
    function cost(p, d0, d1, L0, L1, c)
        if c < zero(c)
            throw(ArgumentError("Cost must be non-negative"))
        end
        x0 = accept_x0(p, L0)
        x1 = accept_x1(p, L1)
        draw = draw_again(p, d0, d1, L0, L1, c, min(x0, x1))
        output = min(abs(draw - x0), abs(draw - x1), abs(x1 - x0))
        return output
    end
    # Find the indifference points
    roots = find_zeros(p - > cost(p, d0, d1, L0, L1, c), 0 + eps(), 1 - eps())
    # Compute the choice at both sides
    left = first.(choice.(roots - eps(), d0, d1, L0, L1, c))
    right = first.(choice.(roots + eps(), d0, d1, L0, L1, c))
    # Find $\beta$ by checking for a permanent transition from the area $\bullet$ to never again accepting $\bullet$ at the various indifference points
    # Find $\alpha$ by checking for a permanent transition from the area $\bullet$ to never again accepting $\bullet$ at the various indifference points
    $\beta$ = findlast((left == 2) & (right ≠ 2)) |> (x -> isa(x, Int) ? x : 0)
    $\alpha$ = findfirst((left ≠ 1) & (right == 1)) |> (x -> isa(x, Int) ? x : 1)
    if $\beta < \alpha$
        @printf("Accept x1 if p ≤ %.2f\nContinue to draw if %.2f ≤ p ≤ %.2f
Accept x0 if p ≥ %.2f", $\beta$, $\beta$, $\alpha$, $\alpha")
    else
        x0 = accept_x0($\beta$, L0)
        x1 = accept_x1($\beta$, L1)
        draw = draw_again($\beta$, d0, d1, L0, L1, c, min(x0, x1))
        if draw == min(x0, x1, draw)
            @printf("Accept x1 if p ≤ %.2f\nContinue to draw if %.2f ≤ p ≤ %.2f
Accept x0 if p ≥ %.2f", $\beta$, $\beta$, $\alpha$, $\alpha")
        else
            @printf("Accept x1 if p ≤ %.2f\nAccept x0 if p ≥ %.2f", $\beta$, $\alpha")
        end
    end
    return ($\alpha$, $\beta$)
end

Out[5]: decision_rule (generic function with 1 method)

We can simulate an agent facing a problem and the outcome with the following function

In [6]: function simulation(problem)
    @unpack d0, d1, L0, L1, c, p, n, return_output = problem
    $\alpha$, $\beta$ = decision_rule(d0, d1, L0, L1, c)
outcomes = fill(false, n)
costs = fill(0.0, n)
trials = fill(0, n)
for trial in 1:n
    # Nature chooses
    truth = rand(1:2)
    # The true distribution and loss are defined based on the truth
d = (d0, d1)[truth]
l = (L0, L1)[truth]
t = 0
choice = 0
while iszero(choice)
    t += 1
    outcome = rand(d)
    p = bayes_update(p, d0, d1)
    if p <= β
        choice = 1
    elseif p >= α
        choice = 2
    end
end
correct = choice == truth
cost = t * c + (correct ? 0 : l)
outcomes[trial] = correct
costs[trial] = cost
trials[trial] = t
end
@printf("\nCorrect: %.2f\nAverage Cost: %.2f\nAverage number of trials: %.2f",
mean(outcomes), mean(costs), mean(trials))
return return_output ? (α, β, outcomes, costs, trials) : nothing
end
Problem = @with_kw (d0 = Beta(1,1), d1 = Beta(9,9),
L0 = 2, L1 = 2,
c = 0.2, p = 0.5,
n = 100, return_output = false);

In [7]: Random.seed!(0);
simulation(Problem());

Accept x1 if p ≤ 0.35
Continue to draw if 0.35 ≤ p ≤ 0.57
Accept x0 if p ≥ 0.57
Correct: 0.43
Average Cost: 1.42
Average number of trials: 1.40

27.5.1 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?

In [8]: Random.seed!(0);
simulation(Problem(c = 0.4));

    Accept x1 if p ≤ 0.41
Continue to draw if 0.41 ≤ p ≤ 0.54
    Accept x0 if p ≥ 0.54
Correct: 0.45
Average Cost: 1.59
Average number of trials: 1.22

Notice what happens?
The average number of trials decreased.

Increased cost per draw has induced the decision maker to decide in 0.18 less trials on average.

Because he decides with less experience, 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.

27.6 Comparison with Neyman-Pearson formulation

For several reasons, it is useful to describe the theory underlying the test that Navy Captain G. S. Schuyler had been told to use and that led him to approach Milton Friedman and Allan Wallis to convey his conjecture that superior practical procedures existed.

Evidently, the Navy had told Captain Schuyler to use what it knew to be a state-of-the-art Neyman-Pearson test.

We’ll rely on Abraham Wald’s [106] elegant summary of Neyman-Pearson theory.

For our purposes, watch for there features of the setup:

• the assumption of a \textit{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 \textit{Sequential Analysis} [106] 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 \textit{something} is already known in order to state a well posed problem. Usually, \textit{something} means \textit{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.

Let’s listen to Wald longer:

- As a basis for choosing among critical regions the following considerations have been advanced by Neyman and Pearson: In accepting or rejecting $H_0$ we may commit errors of two kinds. We commit an error of the first kind if we reject $H_0$ when it is true; we commit an error of the second kind if we accept $H_0$ when $H_1$ is true. After a particular critical region $W$ has been chosen, the probability of committing an error of the first kind, as well as the probability of committing an error of the second kind is uniquely determined. The probability of committing an error of the first kind is equal to the probability, determined by the assumption that $H_0$ is true, that the observed sample will be included in the critical region $W$. The probability of committing an error of the second kind is equal to the probability, determined on the assumption that $H_1$ is true, that the probability will fall outside the critical region $W$. For any given critical region $W$ we shall denote the probability of an error of the first kind by $\alpha$ and the probability of an error of the second kind by $\beta$.

Let’s listen carefully to how Wald applies a law of large numbers to interpret $\alpha$ and $\beta$:

- The probabilities $\alpha$ and $\beta$ have the following important practical interpretation: Suppose that we draw a large number of samples of size $n$. Let $M$ be the number of such samples drawn. Suppose that for each of these $M$ samples we reject $H_0$ if the sample is included in $W$ and accept $H_0$ if the sample lies outside $W$. In this way we make $M$ statements of rejection or acceptance. Some of these statements will in general be wrong. If $H_0$ is true and if $M$ is large, the probability is nearly 1 (i.e., it is practically certain) that the proportion of wrong statements (i.e., the number of wrong statements divided by $M$) will be approximately $\alpha$. If $H_1$ is true, the probability is nearly 1 that the proportion of wrong statements will be approximately $\beta$. Thus, we can say that in the long run [here Wald applies a law of large numbers by driving $M \to \infty$ (our comment, not 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 $m$th trial for each integral value of $m$): (1) to accept the hypothesis $H$, (2) to reject the hypothesis $H$, (3) to continue the experiment by making an additional observation. Thus, such a test procedure is carried out sequentially. On the basis of the first observation one of the aforementioned decisions is made. If the first or second decision is made, the process is terminated. If the third decision is made, a second trial is performed. Again, on the basis of the first two observations one of the three decisions is made. If the third decision is made, a third trial is performed, and so on. The process is continued until either the first or the second decisions is made. The number $n$ of observations required by such a test procedure is a random variable, since the value of $n$ depends on the outcome of the observations.

Footnotes

[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 [62] chapter 11, for a discussion of exchangeability.
Chapter 28

Job Search III: Search with Learning

28.1 Contents

• Overview 28.2
• Model 28.3
• Take 1: Solution by VFI 28.4
• Take 2: A More Efficient Method 28.5
• Exercises 28.6
• Solutions 28.7

28.2 Overview

In this lecture we consider an extension of the previously studied job search model of McCall [76].

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 [68], section 6.6.

28.2.1 Model features

• Infinite horizon dynamic programming with two states and one binary control.
• Bayesian updating to learn the unknown distribution.
28.2.2 Setup

In [1]: `using InstantiateFromURL
   # optionally add arguments to force installation: instantiate = true, precompile = true
   github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")`

28.3 Model

Let’s first review the basic McCall model [76] and then add the variation we want to consider.

28.3.1 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 $\mathbb{E} \sum_{t=0}^{\infty} \beta^t y_t$ The function $V$ satisfies the recursion

$$V(w) = \max \left\{ \frac{w}{1-\beta}, c + \beta \int V(w') h(w') dw' \right\}$$  \hspace{1cm} (1)

The optimal policy has the form $1\{w \geq \tilde{w}\}$, where $\tilde{w}$ is a constant depending called the reservation wage.

28.3.2 Offer Distribution Unknown

Now let’s extend the model by considering the variation presented in [68], section 6.6.

The model is as above, apart from the fact that

- the density $h$ is unknown
- the worker learns about $h$ by starting with a prior and updating based on wage offers that he/she observes

The worker knows there are two possible distributions $F$ and $G$ — with densities $f$ and $g$.

At the start of time, “nature” selects $h$ to be either $f$ or $g$ — the wage distribution from which the entire sequence $\{W_t\}$ will be drawn.

This choice is not observed by the worker, who puts prior probability $\pi_0$ on $f$ being chosen.

Update rule: worker’s time $t$ estimate of the distribution is $\pi_t f + (1 - \pi_t) g$, where $\pi_t$ updates via
\[
\pi_{t+1} = \frac{\pi_t f(w_{t+1})}{\pi_t f(w_{t+1}) + (1 - \pi_t)g(w_{t+1})}
\] (2)

This last expression follows from Bayes’ rule, which tells us that

\[
\mathbb{P}\{h = f \mid W = w\} = \frac{\mathbb{P}\{W = w \mid h = f\}\mathbb{P}\{h = f\}}{\mathbb{P}\{W = w\}} \quad \text{and} \quad \mathbb{P}\{W = w\} = \sum_{\psi \in \{f, g\}} \mathbb{P}\{W = w \mid h = \psi\}\mathbb{P}\{h = \psi\}
\]

The fact that (2) is recursive allows us to progress to a recursive solution method. Letting

\[
h_w(w) := \pi f(w) + (1 - \pi)g(w) \quad \text{and} \quad q(w, \pi) := \frac{\pi f(w)}{\pi f(w) + (1 - \pi)g(w)}
\]

we can express the value function for the unemployed worker recursively as follows

\[
V(w, \pi) = \max\left\{ \frac{w}{1 - \beta}, c + \beta \int V(w', \pi') h_w(w') \, dw' \right\} \quad \text{where} \quad \pi' = q(w', \pi)
\] (3)

Notice that the current guess \( \pi \) is a state variable, since it affects the worker’s perception of probabilities for future rewards.

### 28.3.3 Parameterization

Following section 6.6 of \([68]\), 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

\begin{verbatim}
In [2]: using LinearAlgebra, Statistics
     using Distributions, Plots, QuantEcon, Interpolations, Parameters

     gr(fmt=:png);

     w_max = 2
     x = range(0, w_max, length = 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")

Out[2]:
\end{verbatim}
28.3.4 Looking Forward

What kind of optimal policy might result from (3) and the parameterization specified above? Intuitively, if we accept at $w_a$ and $w_a \leq w_b$, then — all other things being given — we should also accept at $w_b$.

This suggests a policy of accepting whenever $w$ exceeds some threshold value $\bar{w}$.

But $\bar{w}$ should depend on $\pi$ — in fact it should be decreasing in $\pi$ because

- $f$ is a less attractive offer distribution than $g$
- larger $\pi$ means more weight on $f$ and less on $g$

Thus larger $\pi$ depresses the worker’s assessment of her future prospects, and relatively low current offers become more attractive.

**Summary:** We conjecture that the optimal policy is of the form $1\{w \geq \bar{w}(\pi)\}$ for some decreasing function $\bar{w}$.

28.4 Take 1: Solution by VFI

Let’s set about solving the model and see how our results match with our intuition.

We begin by solving via value function iteration (VFI), which is natural but ultimately turns out to be second best.

The code is as follows.

```python
In [3]: # use key word argument
    
function SearchProblem(;β = 0.95, c = 0.6, F_a = 1, F_b = 1,
    G_a = 3, G_b = 1.2, w_max = 2.0,
    w_grid_size = 40, π_grid_size = 40)
```
28.4. TAKE 1: SOLUTION BY VFI

\[ F = \text{Beta}(F_a, F_b) \]
\[ G = \text{Beta}(G_a, G_b) \]

# scaled pdfs
\[ f(x) = \frac{\text{pdf}(F, x/w_{\text{max}})}{w_{\text{max}}} \]
\[ g(x) = \frac{\text{pdf}(G, x/w_{\text{max}})}{w_{\text{max}}} \]

\[ \pi_{\text{min}} = 1e-3 \quad \# \text{avoids instability} \]
\[ \pi_{\text{max}} = 1 - \pi_{\text{min}} \]

\[ w_{\text{grid}} = \text{range}(0, w_{\text{max}}, \text{length} = w_{\text{grid\_size}}) \]
\[ \pi_{\text{grid}} = \text{range}(\pi_{\text{min}}, \pi_{\text{max}}, \text{length} = \pi_{\text{grid\_size}}) \]

\[ \text{nodes, weights} = \text{qnwlege}(21, 0.0, w_{\text{max}}) \]

return \((\beta = \beta, c = c, F = F, G = G, f = f, g = g, n_w = w_{\text{grid\_size}}, w_{\text{max}} = w_{\text{max}}, w_{\text{grid}} = w_{\text{grid}}, n_\pi = \pi_{\text{grid\_size}}, \pi_{\text{min}} = \pi_{\text{min}}, \pi_{\text{max}} = \pi_{\text{max}}, \pi_{\text{grid}} = \pi_{\text{grid}}, \text{quad\_nodes} = \text{nodes}, \text{quad\_weights} = \text{weights}) \]

function q(sp, w, \pi_{\text{val}})

\[ \text{new}_\pi = 1.0 / (1 + ((1 - \pi_{\text{val}}) * sp.g(w)) / (\pi_{\text{val}} * sp.f(w))) \]

# Return new_\pi when in \([\pi_{\text{min}}, \pi_{\text{max}}]\) and else end points
return clamp(new_\pi, sp.\pi_{\text{min}}, sp.\pi_{\text{max}})

end

function T!(sp, v, out; ret_policy = false)

# simplify names
@unpack f, g, \beta, c = sp

\[ \text{nodes, weights} = \text{sp.quad\_nodes, sp.quad\_weights} \]

\[ v = \text{extrapolate(\text{interpolate((sp.w\_grid, sp.\pi\_grid), v, Gridded(Linear())), Flat()))} \]

# set up quadrature nodes/weights
# q\_nodes, q\_weights = qn\_wlege(21, 0.0, sp.w_{\text{max}})

for (w_i, w) in enumerate(sp.w\_grid)

# calculate v1
\[ v1 = w / (1 - \beta) \]

for (\pi_j, _\pi) in enumerate(sp.\pi\_grid)

# calculate v2
integrand(m) = [v[m[i], q.(Ref(sp), m[i], _\pi)] *
\[ (_\pi * f(m[i]) + (1 - _\pi) * g(m[i])) \]
\[ \text{for i in 1:} \]
\[ \text{length(m)}] \]
integral = do_quad(integrand, nodes, weights)

# integral = do_quad(integrand, q\_nodes, q\_weights)

\[ v2 = c + \beta * \text{integral} \]

# return policy if asked for, otherwise return max of values
out[w_i, \pi_j] = ret\_policy ? v1 > v2 : \text{max(v1, v2)}

end
The type `SearchProblem` is used to store parameters and methods needed to compute optimal actions.

The Bellman operator is implemented as the method `T()`, 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.

Here’s the value function:

```
In [4]: # Set up the problem and initial guess, solve by VFI
```
sp = SearchProblem(; w_grid_size=100, π_grid_size=100)

v_init = fill(sp.c / (1 - sp.β), sp.n_w, sp.n_π)

f(x) = T(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.π_grid), v, Gridded(Linear())), Flat())

pf = extrapolate(interpolate((sp.w_grid, sp.π_grid), policy, Gridded(Linear())), Flat())

function plot_value_function(;w_plot_grid_size = 100,
π_plot_grid_size = 100)

π_plot_grid = range(0.001, 0.99, length = π_plot_grid_size)
w_plot_grid = range(0, sp.w_max, length = w_plot_grid_size)

Z = [vf(w_plot_grid[j], π_plot_grid[i])
    for j in 1:w_plot_grid_size, i in 1:π_plot_grid_size]

p = contour(π_plot_grid, w_plot_grid, Z, levels=15, alpha=0.6,
    fill=true, size=(400, 400), c=lightrainbow)

plot!(xlabel="pπ", ylabel="w", xguidefont=font(12))

return p
end

plot_value_function()
The optimal policy:

```python
In [5]: function plot_policy_function(;w_plot_grid_size = 100,
                                  π_plot_grid_size = 100)
    π_plot_grid = range(0.001, 0.99, length = π_plot_grid_size)
    w_plot_grid = range(0, sp.w_max, length = 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:π_plot_grid_size]
    p = contour(π_plot_grid, w_plot_grid, Z, levels=1, alpha=0.6, fill=true,
               size=(400, 400), c=:coolwarm)
    plot!(xlabel="pi", ylabel="wage", xguidefont=font(12), cbar=false)
    annotate!(0.4, 1.0, "reject")
    annotate!(0.7, 1.8, "accept")
    return p
end

plot_policy_function()
```

Out[5]:
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.

28.5 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.
28.5.1 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) have equal value:

\[
\frac{\bar{w}(\pi)}{1 - \beta} = c + \beta \int V(w', \pi') h_{\pi}(w') \, dw'
\]  

(4)

Together, (3) and (4) give

\[
V(w, \pi) = \max \left\{ \frac{w}{1 - \beta}, \frac{\bar{w}(\pi)}{1 - \beta} \right\}
\]

(5)

Combining (4) and (5), we obtain

\[
\frac{\bar{w}(\pi)}{1 - \beta} = c + \beta \int \max \left\{ \frac{w'}{1 - \beta}, \frac{\bar{w}(\pi')}{1 - \beta} \right\} h_{\pi}(w') \, dw'
\]

Multiplying by \( 1 - \beta \), substituting in \( \pi' = q(w', \pi) \) and using \( \circ \) for composition of functions yields

\[
\bar{w}(\pi) = (1 - \beta)c + \beta \int \max \{ w', \bar{w} \circ q(w', \pi) \} h_{\pi}(w') \, dw'
\]

(6)

Equation (6) 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.

28.5.2 Solving the RWFE

To solve the RWFE, we will first show that its solution is the fixed point of a contraction mapping.

To this end, let

- \( b[0,1] \) be the bounded real-valued functions on \([0,1]\)
- \( \|\psi\| := \sup_{x \in [0,1]} |\psi(x)| \)

Consider the operator \( Q \) mapping \( \psi \in b[0,1] \) into \( Q\psi \in b[0,1] \) via

\[
(Q\psi)(\pi) = (1 - \beta)c + \beta \int \max \{ w', \psi \circ q(w', \pi) \} h_{\pi}(w') \, dw'
\]

(7)

Comparing (6) and (7), we see that the set of fixed points of \( Q \) exactly coincides with the set of solutions to the RWFE.

- If \( Q\bar{w} = \bar{w} \) then \( \bar{w} \) solves (6) and vice versa.

Moreover, for any \( \psi, \phi \in b[0,1] \), basic algebra and the triangle inequality for integrals tells us that

\[
| (Q\psi)(\pi) - (Q\phi)(\pi) | \leq \beta \int | \max \{ w', \psi \circ q(w', \pi) \} - \max \{ w', \phi \circ q(w', \pi) \} | h_{\pi}(w') \, dw'
\]

(8)
Working case by case, it is easy to check that for real numbers $a, b, c$ we always have

$$|\max\{a, b\} - \max\{a, c\}| \leq |b - c|$$

Combining (8) and (9) yields

$$|(Q\psi)(\pi) - (Q\phi)(\pi)| \leq \beta \int |\psi \circ q(w', \pi) - \phi \circ q(w', \pi)| h_{\pi}(w') dw' \leq \beta \|\psi - \phi\|$$

Taking the supremum over $\pi$ now gives us

$$\|Q\psi - Q\phi\| \leq \beta \|\psi - \phi\|$$

In other words, $Q$ is a contraction of modulus $\beta$ on the complete metric space $(b[0, 1], \| \cdot \|)$.

Hence

- A unique solution $\tilde{w}$ to the RWFE exists in $b[0, 1]$.
- $Q^k\psi \to \tilde{w}$ uniformly as $k \to \infty$, for any $\psi \in b[0, 1]$.

**Implementation**

These ideas are implemented in the `.res_wage_operator()` method from `odu.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 $\tilde{w}$.

### 28.6 Exercises

#### 28.6.1 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.

### 28.7 Solutions

#### 28.7.1 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`.

In [6]: `sp = SearchProblem(π_grid_size = 50)`
\( \phi_{\text{init}} = \text{ones}(\text{sp}.n_\pi) \)
\( f_{\text{ex1}}(x) = \text{res\_wage\_operator}(\text{sp}, x) \)
\( \hat{w} = \text{compute\_fixed\_point}(f_{\text{ex1}}, \phi_{\text{init}}) \)

```
plot(\text{sp}.\pi\_grid, \hat{w}, \text{linewidth} = 2, \text{color}=:black, 
     \text{fillrange} = \emptyset, \text{fillalpha} = 0.15, \text{fillcolor} = :blue)
plot!(\text{sp}.\pi\_grid, 2 * \text{ones}(\text{length}(\hat{w})), \text{linewidth} = 0, \text{fillrange} = \hat{w},
     \text{fillalpha} = 0.12, \text{fillcolor} = :green, \text{legend} = :none)
plot!(\text{ylims} = (0, 2), \text{annotations} = [(0.42, 1.2, "\text{reject}")],
     (0.7, 1.8, "\text{accept}"))
```

Compute iterate 10 with error 0.00719443760325555
Compute iterate 20 with error 0.0004348703417873523
Converged in 26 steps

The next piece of code is not one of the exercises from QuantEcon – it’s just a fun simulation to see what the effect of a change in the underlying distribution on the unemployment rate is.

At a point in the simulation, the distribution becomes significantly worse. It takes a while for agents to learn this, and in the meantime they are too optimistic, and turn down too many jobs. As a result, the unemployment rate spikes.

The code takes a few minutes to run.

```
In [7]: # Determinism and random objects.
   using Random
   Random.seed!(42)
```
# Set up model and compute the function $\hat{w}$

```julia
sp = SearchProblem(π_grid_size = 50, F_a = 1, F_b = 1)
ϕ_init = ones(sp.n_π)
g(x) = res_wage_operator(sp, x)
\hat{w}_vals = compute_fixed_point(g, ϕ_init)
\hat{w} = extrapolate(interpolate((sp.π_grid, ), \hat{w}_vals, Gridded(Linear())), Flat())
```

# Holds the employment state and beliefs of an individual agent.
```julia
mutable struct Agent{TF <: AbstractFloat, TI <: Integer}
    _π::TF
    employed::TI
end
Agent(_π=1e-3) = Agent(_π, 1)
```

```julia
function update!(ag, H)
    if ag.employed == 0
        w = rand(H) * 2  # account for scale in julia
        if w ≥ \hat{w}(ag._π)
            ag.employed = 1
        else
            ag._π = 1.0 ./ (1 + ((1 - ag._π) .* sp.g(w)) ./ (ag._π * 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 = zeros(sim_length)

for i in 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
```

```julia
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 = "")
28.7. SOLUTIONS
Chapter 29

Job Search IV: Modeling Career Choice

29.1 Contents

- Overview 29.2
- Model 29.3
- Exercises 29.4
- Solutions 29.5

29.2 Overview

Next we study a computational problem concerning career and job choices. The model is originally due to Derek Neal [81]. This exposition draws on the presentation in [68], section 6.5.

29.2.1 Model features

- Career and job within career both chosen to maximize expected discounted wage flow.
- Infinite horizon dynamic programming with two state variables.

29.2.2 Setup

In [1]: using InstantiateFromURL
   # optionally add arguments to force installation: instantiate = true,
   github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

In [2]: using LinearAlgebra, Statistics

29.3 Model

In what follows we distinguish between a career and a job, where
• a career is understood to be a general field encompassing many possible jobs, and
• a job is understood to be a position with a particular firm

For workers, wages can be decomposed into the contribution of job and career

\[ w_t = \theta_t + \epsilon_t, \]

where
- \( \theta_t \) is contribution of career at time \( t \)
- \( \epsilon_t \) is contribution of job at time \( t \)

At the start of time \( t \), a worker has the following options

• retain a current (career, job) pair \((\theta_t, \epsilon_t)\) — referred to hereafter as “stay put”
• retain a current career \( \theta_t \) but redraw a job \( \epsilon_t \) — referred to hereafter as “new job”
• redraw both a career \( \theta_t \) and a job \( \epsilon_t \) — referred to hereafter as “new life”

Draws of \( \theta \) and \( \epsilon \) are independent of each other and past values, with

• \( \theta_t \sim F \)
• \( \epsilon_t \sim G \)

Notice that the worker does not have the option to retain a job but redraw a career — starting a new career always requires starting a new job.

A young worker aims to maximize the expected sum of discounted wages.

\[
\mathbb{E} \sum_{t=0}^{\infty} \beta^t w_t
\]

subject to the choice restrictions specified above.

Let \( V(\theta, \epsilon) \) denote the value function, which is the maximum of (1) over all feasible (career, job) policies, given the initial state \((\theta, \epsilon)\).

The value function obeys

\[
V(\theta, \epsilon) = \max\{I, II, III\},
\]

where

\[
\begin{align*}
I &= \theta + \epsilon + \beta V(\theta, \epsilon) \\
II &= \theta + \int \epsilon' G(\epsilon') + \beta \int V(\theta', \epsilon') G(\epsilon') \, d\epsilon' \\
III &= \int \theta' F(d\theta') + \int \epsilon' G(\epsilon') + \beta \int \int V(\theta', \epsilon') G(\epsilon') F(d\theta')
\end{align*}
\]

Evidently \( I, II \) and \( III \) correspond to “stay put”, “new job” and “new life”, respectively.

### 29.3.1 Parameterization

As in [68], 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 \( \text{linspace}(\theta, B, N) \) — 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 \texttt{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 $n = 50$.

In [3]: using Plots, QuantEcon, Distributions

\begin{verbatim}
gr(fmt=:png);

n = 50
a_vals = [0.5, 1, 100]
b_vals = [0.5, 1, 100]

plt = plot()
for (a, b) in zip(a_vals, b_vals)
    ab_label = "a = $a, b = $b"
    dist = BetaBinomial(n, a, b)
    plot!(plt, 0:n, pdf.(dist, support(dist)), label = ab_label)
end
plt
\end{verbatim}

Out[3]:
Implementation:

The code for solving the DP problem described above is found below:

In [4]: function CareerWorkerProblem(\(\beta = 0.95\),
                                      \(B = 5.0\),
                                      \(N = 50\),
                                      \(F_a = 1.0\),
                                      \(F_b = 1.0\),
                                      \(G_a = 1.0\),
                                      \(G_b = 1.0\))

\[\theta = \text{range}(0, B, \text{length} = N)\]
\[\epsilon = \text{copy}(\theta)\]
\(\text{dist}_F = \text{BetaBinomial}(N-1, F_a, F_b)\)
\(\text{dist}_G = \text{BetaBinomial}(N-1, G_a, G_b)\)
\(\text{F_probs} = \text{pdf}.(\text{dist}_F, \text{support}(\text{dist}_F))\)
\(101x275)
\(\text{G_probs} = \text{pdf}.(\text{dist}_G, \text{support}(\text{dist}_G))\)
\(\text{F_mean} = \text{sum}(\theta . * \text{F_probs})\)
\(\text{G_mean} = \text{sum}(\epsilon . * \text{G_probs})\)

\(\text{return } (\beta = \beta, N = N, B = B, \theta = \theta, \epsilon = \epsilon,\)
\(\text{F_probs} = \text{F_probs}, \text{G_probs} = \text{G_probs},\)
\(\text{F_mean} = \text{F_mean}, \text{G_mean} = \text{G_mean})\)

end

function update_bellman!(cp, v, out; ret_policy = false)

# new life. This is a function of the distribution parameters and is
# always constant. No need to recompute it in the loop
v3 = (cp.G_mean + cp.F_mean .+ cp.\(\beta \) .
    cp.F_probs' * v * cp.G_probs)[1] # do not need 1 element array

for j in 1:cp.N
    for i in 1:cp.N
        # stay put
\[ v_1 = \text{cp.} \theta[i] + \text{cp.} \epsilon[j] + \text{cp.} \beta * v[i, j] \]

# new job
\[ v_2 = (\text{cp.} \theta[i] + \text{cp.} \text{G.mean} + \text{cp.} \beta) \star v[i, :]' * \text{cp.} \text{G.probs}[i] \] # do not need a single element

if ret_policy
    if \( v_1 > \max(v_2, v_3) \)
        action = 1
    elseif \( v_2 > \max(v_1, v_3) \)
        action = 2
    else
        action = 3
    end
    out[i, j] = action
else
    out[i, j] = \( \max(v_1, v_2, v_3) \)
end

function update_bellman(cp, v; ret_policy = false)
    out = similar(v)
    update_bellman!(cp, v, out, ret_policy = ret_policy)
    return out
end

function get_greedy!(cp, v, out)
    update_bellman!(cp, v, out, ret_policy = true)
end

function get_greedy(cp, v)
    update_bellman(cp, v, ret_policy = true)
end

Out[4]: get_greedy (generic function with 1 method)

The code defines

- a named tuple CareerWorkerProblem that
  - encapsulates all the details of a particular parameterization
  - implements the Bellman operator \( T \)

In this model, \( T \) is defined by \( T v(\theta, \epsilon) = \max\{I, II, III\} \), where \( I, II \) and \( III \) are as given in (2), replacing \( V \) with \( v \).

The default probability distributions in CareerWorkerProblem correspond to discrete uniform distributions (see the Beta-binomial figure).

In fact all our default settings correspond to the version studied in [68], 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
In [5]: 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(linetype = :surface, wp.θ, wp.ε, transpose(v), xlabel="theta", ylabel="epsilon",
        seriescolor=:plasma, gridalpha = 1)
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.

### 29.4 Exercises

#### 29.4.1 Exercise 1

Using the default parameterization in the `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`.

29.4.2 Exercise 2

Let’s now consider how long it takes for the worker to settle down to a permanent job, given a starting point of $(\theta, \epsilon) = (0, 0)$.

In other words, we want to study the distribution of the random variable

$$T^* := \text{the first point in time from which the worker’s job no longer changes}$$

Evidently, the worker’s job becomes permanent if and only if $(\theta_t, \epsilon_t)$ 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.

29.4.3 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 the plots functions (e.g., `contour`, `contour!`) to produce the different shadings.

Now set \( G_a = G_b = 100 \) and generate a new figure with these parameters. Interpret.

### 29.5 Solutions

#### 29.5.1 Exercise 1

In [6]: wp = CareerWorkerProblem()

```python
function solve_wp(wp)
    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[]
    ϵ_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
        push!(θ_ind, i)
        push!(ϵ_ind, j)
    end
    return wp.θ[θ_ind], wp.ϵ[ϵ_ind]
end

plot_array = Any[]
for i in 1:2
    θ_path, ϵ_path = gen_path()
    plt = plot(ϵ_path, label="epsilon")
    plot!(plt, θ_path, label="theta")
    plot!(plt, legend=:bottomright)
    push!(plot_array, plt)
end
plot(plot_array..., layout = (2,1))
```

Out[6]:
29.5.2 Exercise 2

The median for the original parameterization can be computed as follows

\[
\text{In [7]: } \text{function} \space \text{gen\_first\_passage\_time(optimal\_policy)} \\
\text{ \quad t} = 0 \\
\text{ \quad i, j} = 1 \\
\text{ \quad while true} \\
\text{ \quad \quad if optimal\_policy[i, j] == 1 \# Stay put} \\
\text{ \quad \quad \quad return t} \\
\text{ \quad \quad elseif optimal\_policy[i, j] == 2 \# New job} \\
\text{ \quad \quad \quad j} = \text{rand}(G)[1] \\
\text{ \quad \quad else \# New life} \\
\text{ \quad \quad \quad i, j} = \text{rand}(F)[1], \text{rand}(G)[1] \\
\text{ \quad \quad end} \\
\text{ \quad t} += 1 \\
\text{ \quad end}
\]

\[
M = 25000 \\
samples = \text{zeros(M)} \\
\text{for i in 1:M} \\
\text{ \quad samples[i]} = \text{gen\_first\_passage\_time(optimal\_policy)} \\
\text{end} \\
\text{print(median(samples))}
\]

7.0

To compute the median with \( \beta = 0.99 \) instead of the default value \( \beta = 0.95 \), replace
\[
\text{wp} = \text{CareerWorkerProblem()} \text{ \text{with} \text{ wp} = \text{CareerWorkerProblem(}\beta=0.99)\text{.}}
\]
The medians are subject to randomness, but should be about 7 and 14 respectively. Not surprisingly, more patient workers will wait longer to settle down to their final job.

In [8]: wp2 = CareerWorkerProblem(\(\beta=0.99\))

\(v_2, \text{optimal\_policy}_2 = \text{solve\_wp}(wp2)\)

\(\text{samples}_2 = \text{zeros}(M)\)

\(\text{for } i \text{ in } 1:M\)
\(\text{samples}_2[i] = \text{gen\_first\_passage\_time}(\text{optimal\_policy}_2)\)
\(\text{end}\)

\(\text{print}(\text{median}(\text{samples}_2))\)

14.0

29.5.3 Exercise 3

Here’s the code to reproduce the original figure

In [9]: wp = CareerWorkerProblem();
   \(v, \text{optimal\_policy} = \text{solve\_wp}(wp)\)

\(\text{lvls} = [0.5, 1.5, 2.5, 3.5]\)
\(\text{x\_grid} = \text{range}(0, 5, \text{length} = 50)\)
\(\text{y\_grid} = \text{range}(0, 5, \text{length} = 50)\)

\(\text{contour(x\_grid, y\_grid, optimal\_policy'), fill=true, levels=lvls, color = :Blues, fillalpha=1, cbar = false}\)
\(\text{contour!(xlabel="theta", ylabel="epsilon")}\)
\(\text{annotate!([(1.8,2.5, text("new\ life", 14, :white, :center))])}\)
\(\text{annotate!([(4.5,2.5, text("new\ job", 14, :center))])}\)
\(\text{annotate!([(4.0,4.5, text("stay\ put", 14, :center))])}\)

Out[9]:
Now, we need only swap out for the new parameters

```python
In [10]: wp = CareerWorkerProblem(G_a=100.0, G_b=100.0); # use new params
    v, optimal_policy = solve_wp(wp)

    lvls = [0.5, 1.5, 2.5, 3.5]
    x_grid = range(0, 5, length = 50)
    y_grid = range(0, 5, length = 50)

    contour(x_grid, y_grid, optimal_policy, fill=true, levels=lvls,color = :Blues,
            fillalpha=1, cbar = false)
    contour!(xlabel="theta", ylabel="epsilon")
    annotate!([(1.8,2.5, text("new life", 14, :white, :center))])
    annotate!([(4.5,2.5, text("new job", 14, :center))])
    annotate!([(4.0,4.5, text("stay put", 14, :center))])

Out[10]:
```

```
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.
Chapter 30

Job Search V: On-the-Job Search

30.1 Contents

- Overview 30.2
- Model 30.3
- Implementation 30.4
- Solving for Policies 30.5
- Exercises 30.6
- Solutions 30.7

30.2 Overview

30.2.1 Model features

- job-specific human capital accumulation combined with on-the-job search
- infinite horizon dynamic programming with one state variable and two controls

30.2.2 Setup

In [1]: using InstantiateFromURL
   # optionally add arguments to force installation: instantiate = true, precompile = true
   github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

In [2]: using LinearAlgebra, Statistics
   using Distributions, QuantEcon, Interpolations, Expectations, Parameters
   using Plots, NLsolve, Random
   gr(fmt = :png);

30.3 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

\[
    x_{t+1} = (1 - b_{t+1})G(x_t, \phi_t) + b_{t+1} \max \{G(x_t, \phi_t), U_{t+1}\} 
\]  

(1)

Agent’s objective: maximize expected discounted sum of wages via controls \( \{s_t\} \) and \( \{\phi_t\} \).

Taking the expectation of \( V(x_{t+1}) \) and using (1), the Bellman equation for this problem can be written as

\[
    V(x) = \max_{s + \phi \leq 1} \left\{ x(1 - s - \phi) + \beta(1 - \pi(s))V[G(x, \phi)] + \beta\pi(s) \int V[G(x, \phi) \lor u]F(du) \right\}. 
\]  

(2)

Here nonnegativity of \( s \) and \( \phi \) is understood, while \( a \lor b := \max\{a, b\} \).

### 30.3.1 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.

### 30.3.2 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 (1) gives expected next period capital equal to $\pi(s)\mathbb{E}U = \mathbb{E}U = 0.5$.
- If $s = 0$ and $\phi = 1$, then next period capital is $G(x, \phi) = G(0.05, 1) \approx 0.23$.

Both rates of return are good, but the return from search is better.

Next suppose that $x = 0.4$

- If $s = 1$ and $\phi = 0$, then expected next period capital is again 0.5
- If $s = 0$ and $\phi = 1$, then $G(x, \phi) = G(0.4, 1) \approx 0.8$

Return from investment via $\phi$ dominates expected return from search.

Combining these observations gives us two informal predictions:

1. At any given state $x$, the two controls $\phi$ and $s$ will function primarily as substitutes — worker will focus on whichever instrument has the higher expected return.

2. For sufficiently small $x$, search will be preferable to investment in job-specific human capital. For larger $x$, the reverse will be true.

Now let's turn to implementation, and see if we can match our predictions.

### 30.4 Implementation

The following code solves the DP problem described above

```python
In [3]: using Distributions, QuantEcon, Interpolations, Expectations, Parameters

# model object
function JvWorker(;A = 1.4, 
    α = 0.6, 
    β = 0.96, 
    grid_size = 50, 
    ϵ = 1e-4)

G(x, φ) = A .* (x .* φ).^α
π_func = sqrt
F = Beta(2, 2)

# expectation operator
E = expectation(F)

# Set up grid over the state space for DP
# Max of grid is the max of a large quantile value for F and the
# fixed point $y = G(y, 1)$.
```
grid_max = max(A^(1.0 / (1.0 - α)), quantile(F, 1 - ϵ))

# range for range(ε, grid_max, grid_size). Needed for # CoordInterpGrid below
x_grid = range(ε, grid_max, length = grid_size)

return (A = A, α = α, β = β, x_grid = x_grid, G = G, π_func = π_func, F = F, E = E, ϵ = ϵ)
end

function T!(jv, V, new_V::AbstractVector)

# simplify notation
@unpack G, π_func, F, β, E, ϵ = jv

# prepare interpoland of value function
Vf = LinearInterpolation(jv.x_grid, V, extrapolation_bc=Line())

# instantiate the linesearch variables
max_val = -1.0
cur_val = 0.0
max_s = 1.0
max_ϕ = 1.0
search_grid = range(ε, 1.0, length = 15)

for (i, x) in enumerate(jv.x_grid)

    function w(z)
        s, ϕ = z
        h(u) = Vf(max(G(x, ϕ), u))
        integral = E(h)
        q = π_func(s) * integral + (1.0 - π_func(s)) * Vf(G(x, ϕ))
        return -x * (1.0 - ϕ - s) - β * q
    end

    for s in search_grid
        for ϕ in search_grid
            cur_val = ifelse(s + ϕ <= 1.0, -w((s, ϕ)), -1.0)
            if cur_val > max_val
                max_val, max_s, max_ϕ = cur_val, s, ϕ
            end
        end
    end

    new_V[i] = max_val
end

function T!(jv, V, out::Tuple{AbstractVector, AbstractVector})

# simplify notation
@unpack G, π_func, F, β, E, ϵ = jv

# prepare interpoland of value function
Vf = LinearInterpolation(jv.x_grid, V, extrapolation_bc=Line())

# instantiate variables
s_policy, ϕ_policy = out[1], out[2]

# instantiate the linesearch variables
max_val = -1.0
cur_val = 0.0
max_s = 1.0
max_ϕ = 1.0
search_grid = range(ε, 1.0, length = 15)

for (i, x) in enumerate(jv.x_grid)
  function w(z)
    s, ϕ = z
    h(u) = Vf(max(G(x, ϕ), u))
    integral = E(h)
    q = π_func(s) * integral + (1.0 - π_func(s)) * Vf(G(x, ϕ))
    return -x * (1.0 - ϕ - s) - β * q
  end

  for s in search_grid
    for ϕ in search_grid
      cur_val = ifelse(s + ϕ <= 1.0, -w((s, ϕ)), -1.0)
      if cur_val > max_val
        max_val, max_s, max_ϕ = cur_val, s, ϕ
      end
    end
  end

  s_policy[i], ϕ_policy[i] = max_s, max_ϕ
end

function T(jv, V; ret_policies = false)
  out = ifelse(ret_policies, (similar(V), similar(V)), similar(V))
  T!(jv, V, out)
  return out
end

Out[3]: T (generic function with 1 method)

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 write a constructor called **JvWorker** that

- packages all the parameters and other basic attributes of a given model
- implements the method \( T \) for value function iteration
The \( T \) method takes a candidate value function \( V \) and updates it to \( TV \) via

\[
TV(x) = - \min_{s+\phi \leq 1} w(s, \phi)
\]

where

\[
w(s, \phi) := - \left\{ x(1-s-\phi) + \beta(1-\pi(s))V[G(x, \phi)] + \beta\pi(s) \int V[G(x, \phi) \lor u]F(du) \right\}
\]

Here we are minimizing instead of maximizing to fit with 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 \( (3) \), 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 \( T \)

- 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 \( (3) \).

The function is minimized over all feasible \( (s, \phi) \) pairs, either by brute-force search over a grid, or specialized solver routines.

The latter is much faster, but convergence to the global optimum is not guaranteed. Grid search is a simple way to check results.

### 30.5 Solving for Policies

Let’s plot the optimal policies and see what they look like.

The code is as follows

```julia
In [4]: wp = JvWorker(grid_size=25)
   v_init = collect(wp.x_grid) .* 0.5
   f(x) = T(wp, x)
   V = fixedpoint(f, v_init)
   sol_V = V.zero
   s_policy, phi_policy = T(wp, sol_V, ret_policies = true)

# plot solution
p = plot(wp.x_grid, [phi_policy s_policy sol_V],
    title = ["phi policy" "s policy" "value function"],
    color = [:orange :blue :green],
    xaxis = ("x", (0.0, maximum(wp.x_grid))),
    yaxis = ((-0.1, 1.1)), size = (800, 800),
```
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.

30.6 Exercises

30.6.1 Exercise 1

Let’s look at the dynamics for the state process $\{x_t\}$ associated with these policies.
The dynamics are given by (1) when $\phi_t$ and $s_t$ are chosen according to the optimal policies, and $\mathbb{P}\{b_{t+1} = 1\} = \pi(s_t)$.

Since the dynamics are random, analysis is a bit subtle.

One way to do it is to plot, for each $x$ in a relatively fine grid called `plot_grid`, a large number $K$ of realizations of $x_{t+1}$ given $x_t = x$. Plot this with one dot for each realization, in the form of a 45 degree diagram. Set

\[
K = 50
\]

\[
\text{plot}\_\text{grid}\_\text{max}, \text{plot}\_\text{grid}\_\text{size} = 1.2, 100
\]

\[
\text{plot}\_\text{grid} = \text{range}(0, \text{plot}\_\text{grid}\_\text{max}, \text{length} = \text{plot}\_\text{grid}\_\text{size})
\]

\[
\text{plot}(\text{plot}\_\text{grid}, \text{plot}\_\text{grid}, \text{color} = \text{:black}, \text{linestyle} = \text{:dash}, \text{lims} = (0, \text{plot}\_\text{grid}\_\text{max}), \text{legend} = \text{:none})
\]

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 $\phi_t \approx 0.6$.

### 30.6.2 Exercise 2

In the preceding exercise we found that $s_t$ converges to zero and $\phi_t$ converges to about 0.6.

Since these results were calculated at a value of $\beta$ close to one, let’s compare them to the best choice for an infinitely patient worker.

Intuitively, an infinitely patient worker would like to maximize steady state wages, which are a function of steady state capital.

You can take it as given—it’s certainly true—that the infinitely patient worker does not search in the long run (i.e., $s_t = 0$ for large $t$).

Thus, given $\phi$, steady state capital is the positive fixed point $x^*(\phi)$ of the map $x \mapsto G(x, \phi)$.

Steady state wages can be written as $w^*(\phi) = x^*(\phi)(1 - \phi)$.

Graph $w^*(\phi)$ with respect to $\phi$, and examine the best choice of $\phi$.

Can you give a rough interpretation for the value that you see?

### 30.7 Solutions

#### 30.7.1 Exercise 1

Here’s code to produce the 45 degree diagram

```python
In [5]: wp = JvWorker(grid_size=25)
    #: simplify notation
    @unpack G, p_func, F = wp

    v_init = collect(wp.x_grid) * 0.5
    f2(x) = T(wp, x)
    V2 = fixedpoint(f2, v_init)
    sol_V2 = V2.zero
```
s_policy, \( \phi_{\text{policy}} = T(wp, \text{sol}_V2, \text{ret}_{\text{policies}}=\text{true}) \)

# Turn the policy function arrays into CoordInterpGrid objects for interpolation
s = LinearInterpolation(wp.x_grid, s_policy, extrapolation_bc=Line())
\( \phi = \text{LinearInterpolation}(wp.x\_\text{grid}, \phi_{\text{policy}}, \text{extrapolation}\_\text{bc}=\text{Line()}) \)

\( h_{\text{func}}(x, b, U) = (1 - b) * G(x, \phi(x)) + b * \max(G(x, \phi(x)), U) \)

Out[5]: \( h_{\text{func}} \) (generic function with 1 method)

In [6]: using Random
Random.seed!(42)
K = 50

plot_grid_max, plot_grid_size = 1.2, 100
plot_grid = range(0, plot_grid_max, length = 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
        U = rand(wp.F)
        y = h_func(x, b, U)
        push!(xs, x)
        push!(ys, y)
    end
end

plot(plot_grid, plot_grid, color=:black, linestyle=:dash, legend=:none)
scatter!(xs, ys, alpha=0.25, color=:green, lims=(0, plot_grid_max),
        ticks=ticks)
plot!(xlabel="x_t", ylabel="x_{t+1}", guidefont=font(16))

Out[6]:
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.

https://julia.quantecon.org/dynamic_programming/jv.html#Solving-for-Policies

we see that $x_t \approx 1$ means that $s_t = s(x_t) \approx 0$ and $\phi_t = \phi(x_t) \approx 0.6$.

### 30.7.2 Exercise 2

In [7]: wp = JvWorker(grid_size=25)

    xbar(\phi) = (wp.A \* \phi^wp.a)^{(1.0 / (1.0 - wp.a))}

    \phi\_grid = range(0, 1, length = 100)

    plot(\phi\_grid, [xbar(\phi) \* (1 - \phi) \textbf{for} \phi \textbf{in} \phi\_grid], color = :blue, 
    label = "w^\phi", legendfont = font(12), xlabel = "\phi", 
    guidefont = font(16), grid = \textbf{false}, legend = :topleft)
Observe that the maximizer is around 0.6.

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.
Chapter 31

Optimal Growth I: The Stochastic Optimal Growth Model

31.1 Contents

- Overview 31.2
- The Model 31.3
- Computation 31.4
- Exercises 31.5
- Solutions 31.6

31.2 Overview

In this lecture we’re going to study a simple optimal growth model with one agent.

The model is a version of the standard one sector infinite horizon growth model studied in

- [100], chapter 2
- [68], section 3.1
- EDTC, chapter 1
- [102], chapter 12

The technique we use to solve the model is dynamic programming.

Our treatment of dynamic programming follows on from earlier treatments in our lectures on shortest paths and job search.

We’ll discuss some of the technical details of dynamic programming as we go along.

31.3 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.

### 31.3.1 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.

### 31.3.2 Optimization

Taking $y_0$ as given, the agent wishes to maximize

$$\mathbb{E} \left[ \sum_{t=0}^{\infty} \beta^t u(c_t) \right]$$

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) we are assuming that the resource constraint (1) holds with equality — which is reasonable because $u$ is strictly increasing and no output will be wasted at the optimum.

In summary, the agent’s aim is to select a path $c_0, c_1, c_2, \ldots$ for consumption that is
1. nonnegative,

2. feasible in the sense of (1),

3. optimal, in the sense that it maximizes (2) 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.

### 31.3.3 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 [100] (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

$$c_t = \sigma(y_t) \quad \text{for all } t$$

In what follows, we will call $\sigma$ a feasible consumption policy if it satisfies

$$0 \leq \sigma(y) \leq y \quad \text{for all } y \in \mathbb{R}_+ \quad (4)$$

In other words, a feasible consumption policy is a Markov policy that respects the resource constraint.

The set of all feasible consumption policies will be denoted by $\Sigma$.

Each $\sigma \in \Sigma$ determines a continuous state Markov process $\{y_t\}$ for output via

$$y_{t+1} = f(y_t - \sigma(y_t))\xi_{t+1}, \quad y_0 \text{ given} \quad (5)$$

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

\[ \mathbb{E} \left[ \sum_{t=0}^{\infty} \beta^t u(c_t) \right] = \mathbb{E} \left[ \sum_{t=0}^{\infty} \beta^t u(\sigma(y_t)) \right] \]  

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.

### 31.3.4 Optimality

The **policy value function** \( v_\sigma \) associated with a given policy \( \sigma \) is the mapping defined by

\[ v_\sigma(y) = \mathbb{E} \left[ \sum_{t=0}^{\infty} \beta^t u(\sigma(y_t)) \right] \]  

when \( \{y_t\} \) is given by (5) 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

\[ v^*(y) := \sup_{\sigma \in \Sigma} v_\sigma(y) \]  

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 (8) for all \( y \in \mathbb{R}_+ \).

### 31.3.5 The Bellman Equation

With our assumptions on utility and production function, the value function as defined in (8) also satisfies a **Bellman equation**.

For this problem, the Bellman equation takes the form

\[ w(y) = \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int w(f(y - c)z)\phi(dz) \right\} \]  

\( y \in \mathbb{R}_+ \)

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, (9) 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.

### 31.3.6 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

\[
\max_{0 \leq c \leq y} \left\{ u(c) + \beta \int w(f(y - c)z)\phi(dz) \right\}
\]

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

\[
\text{A feasible consumption policy is optimal if and only if it is } v^*-\text{greedy}
\]

The intuition is similar to the intuition for the Bellman equation, which was provided after (9).

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.

### 31.3.7 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

\[
Tw(y) := \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int w(f(y - c)z)\phi(dz) \right\} \quad (y \in \mathbb{R}_+)
\]
In other words, $T$ sends the function $w$ into the new function $Tw$ defined (11).

By construction, the set of solutions to the Bellman equation (9) exactly coincides with the set of fixed points of $T$.

For example, if $Tw = w$, then, for any $y \geq 0$,

$$w(y) = Tw(y) = \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int v^*(f(y-c)z)\phi(dz) \right\}$$

which says precisely that $w$ is a solution to the Bellman equation.

It follows that $v^*$ is a fixed point of $T$.

### 31.3.8 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.

### 31.3.9 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, [61] or [74].
31.4 Computation

Let’s now look at computing the value function and the optimal policy.

31.4.1 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 (11), 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 (11).

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 \( y_1 < y_2 < \cdots < y_J \) and reconstruct it from this information when required.

More precisely, the algorithm will be

1. Begin with an array of values \( \{w_1, \ldots, w_J\} \) representing the values of some initial function \( w \) on the grid points \( \{y_1, \ldots, y_J\} \).
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 (11).
4. Unless some stopping condition is satisfied, set \( \{w_1, \ldots, w_J\} = \{T\hat{w}(y_1), \ldots, T\hat{w}(y_J)\} \) and go to step 2.

How should we go about step 2?

This is a problem of function approximation, and there are many ways to approach it.

What’s important here is that the function approximation scheme must not only produce a good approximation to \( Tw \), but also combine well with the broader iteration algorithm described above.

One good choice from both respects is continuous piecewise linear interpolation (see this paper for further discussion).

The next figure illustrates piecewise linear interpolation of an arbitrary function on grid points 0, 0.2, 0.4, 0.6, 0.8, 1.
31.4.2 Setup

In [1]: using InstantiateFromURL
    # optionally add arguments to force installation: instantiate = true,
    # precompile = true
    github_project("Quantecon/quantecon-notebooks-julia", version = "0.8.0")

In [2]: using LinearAlgebra, Statistics
    using PLOTS, QuantEcon, Interpolations, NLsolve, Optim, Random
    gr(fmt = :png);

In [3]: f(x) = 2.*cos.(6x) .+ sin.(14x) .+ 2.5
    c_grid = 0:.2:1
    f_grid = range(0, 1, length = 150)

    Af = LinearInterpolation(c_grid, f(c_grid))

    plt = plot(xlim = (0,1), ylim = (0,6))
    plot!(plt, f, f_grid, color = :blue, lw = 2, alpha = 0.8, label = "true function")
    plot!(plt, f_grid, Af.(f_grid), color = :green, lw = 2, alpha = 0.8, label = "linear approximation")
    plot!(plt, c_grid, seriestype = :sticks, linestyle = :dash, linewidth = 0.2, alpha = 0.5, label = "")
    plot!(plt, legend = :top)

Out[3]:

Another advantage of piecewise linear interpolation is that it preserves useful shape properties such as monotonicity and concavity / convexity.
31.4.3 The Bellman Operator

Here’s a function that implements the Bellman operator using linear interpolation

```
function T(w, grid, β, u, f, shocks; compute_policy = false)
    w_func = LinearInterpolation(grid, w)
    # objective for each grid point
    objectives = (c -> u(c) + β * mean(w_func.(f(y - c) .* shocks)) for y in grid)
    results = maximize(objectives, 1e-10, grid) # solver result for each grid point
    Tw = Optim.maximum(results) # solver result for each grid point
    if compute_policy
        σ = Optim.maximizer(results)
        return Tw, σ
    end
    return Tw
end
```

Out[4]: T (generic function with 1 method)

Notice that the expectation in (11) 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., [83])

31.4.4 An Example

Let’s test out our operator when

- \(f(k) = k^\alpha\)
- \(u(c) = \ln c\)
- \(\phi\) is the distribution of \(\exp(\mu + \sigma \zeta)\) when \(\zeta\) is standard normal

As is well-known (see [68], section 3.1.2), for this particular problem an exact analytical solution is available, with

\[
v^*(y) = \frac{\ln(1 - \alpha \beta)}{1 - \beta} + \frac{(\mu + \alpha \ln(\alpha \beta))}{1 - \alpha} \left[ \frac{1}{1 - \beta} - \frac{1}{1 - \alpha \beta} \right] + \frac{1}{1 - \alpha \beta} \ln y \quad (12)
\]

The optimal consumption policy is
\[ \sigma^*(y) = (1 - \alpha \beta) y \]

Let’s code this up now so we can test against it below

In [5]:
\[
\begin{align*}
\alpha &= 0.4 \\
\beta &= 0.96 \\
\mu &= 0 \\
\sigma &= 0.1 \\
\end{align*}
\]

\[
\begin{align*}
c1 &= \log(1 - \alpha \beta) / (1 - \beta) \\
c2 &= (\mu + \alpha \log(\alpha \beta)) / (1 - \alpha) \\
c3 &= 1 / (1 - \beta) \\
c4 &= 1 / (1 - \alpha \beta) \\
\end{align*}
\]

# Utility
\[ u(c) = \log(c) \]
\[ \partial u / \partial c(c) = 1 / c \]

# Deterministic part of production function
\[ f(k) = k^\alpha \]
\[ f'(k) = \alpha \cdot k^{(\alpha - 1)} \]

# True optimal policy
\[ c_{\text{star}}(y) = (1 - \alpha \beta) * y \]

# True value function
\[ v_{\text{star}}(y) = c1 + c2 \cdot (c3 - c4) + c4 \cdot \log(y) \]

Out[5]: v_star (generic function with 1 method)

31.4.5 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.

In [6]:
\[
\begin{align*}
\text{using Random} \\
\text{Random.seed!}(42) & \quad \# \text{For reproducible results.} \\
\text{grid\_max} &= 4 & \# \text{Largest grid point} \\
\text{grid\_size} &= 200 & \# \text{Number of grid points} \\
\text{shock\_size} &= 250 & \# \text{Number of shock draws in Monte Carlo integral} \\
\end{align*}
\]

\[
\begin{align*}
\text{grid\_y} &= \text{range}(1e-5, \text{grid\_max}, \text{length} = \text{grid\_size}) \\
\text{shocks} &= \exp(\mu + \sigma \cdot \text{randn}(\text{shock\_size})) \\
\end{align*}
\]

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.

In [7]: \( w = T(v_{\text{star}}(\text{grid}_y), \text{grid}_y, \beta, \log, k \to k^\alpha, \text{shocks}) \)

\[
\begin{align*}
\text{plt} & = \text{plot}(\text{ylim} = (-35, -24)) \\
\text{plot!}(\text{plt}, \text{grid}_y, w, \text{linewidth} = 2, \alpha = 0.6, \text{label} = "T(v_{\text{star}})") \\
\text{plot!}(\text{plt}, v_{\text{star}}, \text{grid}_y, \text{linewidth} = 2, \alpha = 0.6, \text{label} = "v_{\text{star}}") \\
\text{plot!}(\text{plt}, \text{legend} = \text{:bottomright})
\end{align*}
\]

Out[7]:

The two functions are essentially indistinguishable, so we are off to a good start.

Now let’s have a look at iterating with the Bellman operator, starting off from an arbitrary initial condition.

The initial condition we’ll start with is \( w(y) = 5 \ln(y) \)

In [8]: \( w = 5 * \log.(\text{grid}_y) \) # An initial condition -- fairly arbitrary

\[
\begin{align*}
n & = 35 \\
\text{plot}(\text{xlim} = (\text{extrema}(\text{grid}_y)), \text{ylim} = (-50, 10)) \\
\text{lb} & = \text{"initial condition"} \\
\text{plt} & = \text{plot}(\text{grid}_y, w, \text{color} = :black, \text{linewidth} = 2, \alpha = 0.8, \text{label} = \text{lb}) \\
\text{for} \ i \ \text{in} \ 1:n \\
\quad w & = T(w, \text{grid}_y, \beta, \log, k \to k^\alpha, \text{shocks}) \\
\quad \text{plot!}(\text{grid}_y, w, \text{color} = \text{RGBA}(i/n, 0, 1 - i/n, 0.8), \text{linewidth} = 2, \alpha = 0.6, \text{label} = \"\"
\end{align*}
\]

\[
\begin{align*}
\text{lb} & = \text{"true value function"}
\end{align*}
\]
CHAPTER 31. OPTIMAL GROWTH I: THE STOCHASTIC OPTIMAL GROWTH MODEL

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 computes the exact fixed point

In [9]: function solve_optgrowth(initial_w; tol = 1e-6, max_iter = 500)
   fixedpoint(w -> T(w, grid_y, β, u, f, shocks), initial_w).zero # gets
   end

Out[9]: solve_optgrowth (generic function with 1 method)

We can check our result by plotting it against the true value

In [10]: initial_w = 5 * log.(grid_y)
v_star_approx = solve_optgrowth(initial_w)
31.4. 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 \( \sigma(y) = (1 - \alpha \beta)y \).

\[
\text{In [11]: } Tw, \sigma = T(v\_\text{star\_approx}, \text{grid}\_y, \beta, \log, k \to k^\alpha, \text{shocks}; \text{compute\_policy} = \text{true}) \\
c\star = (1 - \alpha \beta) \times \text{grid}\_y
\]

\[
\text{In [12]: } \text{plt} = \text{plot(}\text{grid}\_y, \sigma, \text{lw=2, alpha=0.6, label = "approximate policy\textsuperscript{\textregistered}\ function"}\text{)} \\
\text{plt!}(\text{plt, grid}\_y, c\star, \text{lw = 2, alpha = 0.6, label = "true policy\textsuperscript{\textregistered}\ function"}) \\
\text{plt!(plt, legend = :bottomright)}
\]

Out[11]:

The figure shows that we are pretty much on the money.
The figure shows that we’ve done a good job in this instance of approximating the true policy.

31.5 Exercises

31.5.1 Exercise 1

Once an optimal consumption policy $\sigma$ is given, income follows (5).

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.

```python
s = 0.05
shocks = \exp(\mu + s \cdot \text{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.

### 31.6 Solutions

#### 31.6.1 Exercise 1

Here’s one solution (assuming as usual that you’ve executed everything above)

```python
function simulate_og(\sigma, y0 = 0.1, ts\_length=100)
    y = zeros(ts\_length)
    \xi = \text{randn(ts\_length-1)}
    y[1] = y0
    for t in 1:(ts\_length-1)
        y[t+1] = (y[t] - \sigma(y[t]))^{\alpha} \cdot \exp(\mu + s \cdot \xi[t])
    end
    return y
end
```

```python
plt = plot()
```
for β in (0.9, 0.94, 0.98)
    Tw = similar(grid_y)
    initial_w = 5 * log.(grid_y)

    v_star_approx = fixedpoint(w -> T(w, grid_y, β, u, f, shocks),
                               initial_w).zero
    Tw, σ = T(v_star_approx, grid_y, β, log, k -> k^α, shocks,
              compute_policy = true)

    σ_func = LinearInterpolation(grid_y, σ)
    y = simulate_og(σ_func)

    plot!(plt, y, lw = 2, alpha = 0.6, label = label = "beta = $β")
end

plot!(plt, legend = :bottomright)
Chapter 32

Optimal Growth II: Time Iteration

32.1 Contents

- Overview 32.2
- The Euler Equation 32.3
- Comparison with Value Function Iteration 32.4
- Implementation 32.5
- Exercises 32.6
- Solutions 32.7

32.2 Overview

In this lecture we’ll continue our earlier study of the stochastic optimal growth model.

In that lecture we solved the associated discounted dynamic programming problem using value function iteration.

The beauty of this technique is its broad applicability.

With numerical problems, however, we can often attain higher efficiency in specific applications by deriving methods that are carefully tailored to the application at hand.

The stochastic optimal growth model has plenty of structure to exploit for this purpose, especially when we adopt some concavity and smoothness assumptions over primitives.

We’ll use this structure to obtain an Euler equation based method that’s more efficient than value function iteration for this and some other closely related applications.

In a subsequent lecture we’ll see that the numerical implementation part of the Euler equation method can be further adjusted to obtain even more efficiency.

32.3 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

\[
    v^*(y) = \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int v^*(f(y - c)z) \phi(dz) \right\} \quad \text{for all } y \in \mathbb{R}_+ \quad (1)
\]

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 \((1)\).

The conditions above imply that
- \( c^* \) is the unique optimal policy for the stochastic optimal growth model
- the optimal policy is continuous, strictly increasing and also **interior**, in the sense that \( 0 < c^*(y) < y \) for all strictly positive \( y \), and
- the value function is strictly concave and continuously differentiable, with

\[
    (v^*)'(y) = u'(c^*(y)) := (u' \circ c^*)(y) \quad (2)
\]

The last result is called the **envelope condition** due to its relationship with the envelope theorem.

To see why \((2)\) might be valid, write the Bellman equation in the equivalent form

\[
    v^*(y) = \max_{0 \leq k \leq y} \left\{ u(y - k) + \beta \int v^*(f(k)z) \phi(dz) \right\},
\]
differentiate naively with respect to \( y \), and then evaluate at the optimum.

Section 12.1 of **EDTC** contains full proofs of these results, and closely related discussions can be found in many other texts.

Differentiability of the value function and interiority of the optimal policy imply that optimal consumption satisfies the first order condition associated with \((1)\), which is

\[
    u'(c^*(y)) = \beta \int (v^*)'(f(y - c^*(y))z)f'(y - c^*(y))z \phi(dz) \quad (3)
\]

Combining \((2)\) and the first-order condition \((3)\) gives the famous **Euler equation**

\[
    (u' \circ c^*)(y) = \beta \int (u' \circ c^*)(f(y - c^*(y))z)f'(y - c^*(y))z \phi(dz) \quad (4)
\]

We can think of the Euler equation as a functional equation

\[
    (u' \circ \sigma)(y) = \beta \int (u' \circ \sigma)(f(y - \sigma(y))z)f'(y - \sigma(y))z \phi(dz) \quad (5)
\]

over interior consumption policies \( \sigma \), one solution of which is the optimal policy \( c^* \).

Our aim is to solve the functional equation \((5)\) and hence obtain \( c^* \).
32.3. THE EULER EQUATION

32.3.1 The Coleman Operator

Recall the Bellman operator

\[ T w(y) := \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int w(f(y - c)z)\phi(dz) \right\} \]  

(6)

Just as we introduced the Bellman operator to solve the Bellman equation, we will now introduce an operator over policies to help us solve the Euler equation.

This operator \( K \) will act on the set of all \( \sigma \in \Sigma \) that are continuous, strictly increasing and interior (i.e., \( 0 < \sigma(y) < y \) for all strictly positive \( y \)).

Henceforth we denote this set of policies by \( \mathcal{P} \).

1. The operator \( K \) takes as its argument a \( \sigma \in \mathcal{P} \) and
2. returns a new function \( K\sigma \), where \( K\sigma(y) \) is the \( c \in (0, y) \) that solves

\[ u'(c) = \beta \int (u' \circ \sigma)(f(y - c)z)f'(y - c)z\phi(dz) \]  

(7)

We call this operator the Coleman operator to acknowledge the work of [17] (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 (5).

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 \( c^*(y) \).

32.3.2 Is the Coleman Operator Well Defined?

In particular, is there always a unique \( c \in (0, y) \) that solves (7)?

The answer is yes, under our assumptions.

For any \( \sigma \in \mathcal{P} \), the right side of (7)

- is continuous and strictly increasing in \( c \) on \( (0, y) \)
- diverges to \( +\infty \) as \( c \uparrow y \)

The left side of (7)

- 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 \( \sigma \in \mathcal{P} \).

### 32.4 Comparison with Value Function Iteration

How does Euler equation time iteration compare with value function iteration?

Both can be used to compute the optimal policy, but is one faster or more accurate?

There are two parts to this story.

First, on a theoretical level, the two methods are essentially isomorphic.

In particular, they converge at the same rate.

We’ll prove this in just a moment.

The other side to the story is the speed of the numerical implementation.

It turns out that, once we actually implement these two routines, time iteration is faster and more accurate than value function iteration.

More on this below.

#### 32.4.1 Equivalent Dynamics

Let’s talk about the theory first.

To explain the connection between the two algorithms, it helps to understand the notion of equivalent dynamics.

(This concept is very helpful in many other contexts as well).

Suppose that we have a function \( g : X \rightarrow X \) 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’s the picture

\[
    x_0 \xrightarrow{g} g(x_0) \xrightarrow{g} g^2(x_0) \xrightarrow{g} g^3(x_0) \xrightarrow{g} \cdots
\]

Now let another function \( h : Y \rightarrow Y \) 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

$$g = \tau^{-1} \circ h \circ \tau \quad (8)$$

Here’s a commutative diagram that illustrates

\[
\begin{array}{ccc}
X & \xrightarrow{g} & X \\
\downarrow{\tau} & & \uparrow{\tau^{-1}} \\
Y & \xrightarrow{h} & Y
\end{array}
\]

Here’s a similar figure that traces out the action of the maps on a point $x \in X$

\[
\begin{array}{ccc}
\tau(x) & \xrightarrow{g} & g(\tau(x)) \\
\downarrow{\tau} & & \uparrow{\tau^{-1}} \\
\tau(x) & \xrightarrow{h} & h(\tau(x))
\end{array}
\]

Now, it’s easy to check from (8) that $g^2 = \tau^{-1} \circ h^2 \circ \tau$ holds.
In fact, if you like proofs by induction, you won’t have trouble showing that

\[ g^n = \tau^{-1} \circ h^n \circ \tau \]

is valid for all \( n \).

What does this tell us?

It tells us that the following are equivalent

- iterate \( n \) times with \( g \), starting at \( x \)
- shift \( x \) to \( Y \) using \( \tau \), 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.

### 32.4.2 Back to Economics

Have you guessed where this is leading?

What we’re going to show now is that the operators \( T \) and \( K \) commute under a certain bijection.

The implication is that they have exactly the same rate of convergence.

To make life a little easier, we’ll assume in the following analysis (although not always in our applications) that \( u(0) = 0 \).

#### A Bijection

Let \( \mathcal{V} \) 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

\[ M \circ T = K \circ M \]  

(9)

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!

### 32.5 Implementation

We’ve just shown that the operators \( T \) and \( K \) have the same rate of convergence. However, it turns out that, once numerical approximation is taken into account, significant differences arises.

In particular, the image of policy functions under \( K \) can be calculated faster and with greater accuracy than the image of value functions under \( T \).

Our intuition for this result is that

- the Coleman operator exploits more information because it uses first order and envelope conditions
- policy functions generally have less curvature than value functions, and hence admit more accurate approximations based on grid point information

#### 32.5.1 The Operator

Here’s some code that implements the Coleman operator.

#### 32.5.2 Setup

```julia
In [1]: using InstantiateFromURL
   \# optionally add arguments to force installation: instantiate = true, \n   \# precompile = true
   \github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

In [2]: using LinearAlgebra, Statistics
   \using BenchmarkTools, Interpolations, Parameters, Plots, QuantEcon, Roots
   \using Optim, Random

In [3]: using BenchmarkTools, Interpolations, Parameters, Plots, QuantEcon, Roots
   gr(fmt = :png);

In [4]: function K!(Kg, g, grid, \( \beta \), \( \partial u / \partial c \), f, f', shocks)
   \# This function requires the container of the output value as argument Kg
   \# Construct linear interpolation object
   g_func = LinearInterpolation(grid, g, extrapolation_bc=Line())

   \# solve for updated consumption value
   for (i, y) in enumerate(grid)
       function h(c)
           vals = \( \partial u / \partial c \).((g_func.(f(y - c) .* shocks)).* f'(y - c) .* shocks
           return \( \partial u / \partial c \)(c) - \beta \cdot \text{mean(vals)}
       end
       end
   end
```
\[
K_g[i] = \text{find_zero}(h, (1e-10, y - 1e-10))
\]

end

return Kg
end

# The following function does NOT require the container of the output
\[\text{value as argument}\]
\[
K(g, \text{grid, } \beta, \partial u/\partial c, f, f', \text{shocks}) = \text{K!}(\text{similar}(g), g, \text{grid, } \beta, \partial u/\partial c, f, f', \text{shocks})
\]

Out[4]: K (generic function with 1 method)

It has some similarities to the code for the Bellman operator in our optimal growth lecture. For example, it evaluates integrals by Monte Carlo and approximates functions using linear interpolation.

Here’s that Bellman operator code again, which needs to be executed because we’ll use it in some tests below

In [5]: using Optim

function T(w, grid, \beta, u, f, shocks, Tw = similar(w); compute_policy = false)

    # apply linear interpolation to w
    w_func = LinearInterpolation(grid, w, extrapolation_bc=Line())

    if compute_policy
        \sigma = 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 = maximize(objective, 1e-10, y)

        if compute_policy
            \sigma[i] = Optim.maximizer(res)
        end

        Tw[i] = Optim.maximum(res)
    end

    if compute_policy
        return Tw, \sigma
    else
        return Tw
    end
end

Out[5]: T (generic function with 2 methods)

32.5.3 Testing on the Log / Cobb–Douglas case

As we did for value function iteration, let’s start by testing our method in the presence of a model that does have an analytical solution.
Here’s an object containing data from the log-linear growth model we used in the value function iteration lecture.

In [6]: isoelastic(c, γ) = isone(γ) ? log(c) : (c^(1 - γ) - 1) / (1 - γ)

Model = @with_kw (α = 0.65,  # Productivity parameter
 β = 0.95,  # Discount factor
 γ = 1.0,  # Risk aversion
 μ = 0.0,  # First parameter in
 lognorm(μ, σ)
 s = 0.1,  # Second parameter in
 lognorm(μ, σ)

grid = range(1e-6, 4, length = 200),  # Grid
grid_min = 1e-6,  # Smallest grid point
grid_max = 4.0,  # Largest grid point
grid_size = 200,  # Number of grid points
u = (c, γ = γ) -> isoelastic(c, γ),  # utility function
∂u∂c = c -> c^(-γ),  # u'
f = k -> k^α,  # production function
f' = k -> α * k^(α - 1),  # f'
)

Out[6]: ##NamedTuple_kw#253 (generic function with 2 methods)

Next we generate an instance

In [7]: m = Model();

We also need some shock draws for Monte Carlo integration

In [8]: using Random
Random.seed!(42)  # for reproducible results.

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, let’s see if \( Kc^\star = c^\star \), as implied by the theory

In [9]: function verify_true_policy(m, shocks, c_star)
   # compute (Kc_star)
   @unpack grid, β, ∂u∂c, f, f' = m
   c_star_new = K(c_star, grid, β, ∂u∂c, f, f', shocks)
   
   # plot c_star and Kc_star
   plot(grid, c_star, label = "optimal policy cc_star")
   plot!(grid, c_star_new, label = "Kc_star")
   plot!(legend = :topleft)
end

Out[9]: verify_true_policy (generic function with 1 method)

In [10]: c_star = (1 - m.α * m.β) * m.grid  # true policy (c_star)
   verify_true_policy(m, shocks, c_star)
We can’t really distinguish the two plots, so we are looking good, at least for this test.

Next let’s try iterating from an arbitrary initial condition and see if we converge towards $c^*$. The initial condition we’ll use is the one that eats the whole pie: $c(y) = y$

```
In [11]: function check_convergence(m, shocks, c_star, g_init; n_iter = 15)
    @unpack grid, β, ∂u∂c, f, f' = m
    g = g_init;
    plot!(m.grid, g, lw = 2, alpha = 0.6, label = "initial condition c(y) = y")
    for i in 1:n_iter
        new_g = K(g, grid, β, ∂u∂c, f, f', shocks)
        g = new_g
        plot!(grid, g, lw = 2, alpha = 0.6, label = "")
    end
    plot!(grid, c_star, color = :black, lw = 2, alpha = 0.8,
          label = "true policy function c_star")
    plot!(legend = :topleft)
end
```

Out[11]: `check_convergence (generic function with 1 method)`

```
In [12]: check_convergence(m, shocks, c_star, m.grid, n_iter = 15)
```

Out[12]:
We see that the policy has converged nicely, in only a few steps.

Now let’s compare the accuracy of iteration using the Coleman and Bellman operators.

We’ll generate

1. $K^n c$ where $c(y) = y$
2. $(M \circ T^n \circ M^{-1}) c$ where $c(y) = y$

In each case we’ll compare the resulting policy to $c^*$.

The theory on equivalent dynamics says we will get the same policy function and hence the same errors.

But in fact we expect the first method to be more accurate for reasons discussed above.

In [13]: function iterate_updating(func, arg_init; sim_length = 20)
    arg = arg_init;
    for i in 1:sim_length
        new_arg = func(arg)
        arg = new_arg
    end
    return arg
end

function compare_error(m, shocks, g_init, w_init; sim_length = 20)
    @unpack grid, β, u, ∂u∂c, f, f′ = m
    g, w = g_init, w_init

    # two functions for simplification
    bellman_single_arg(w) = T(w, grid, β, u, f, shocks)
    coleman_single_arg(g) = K(g, grid, β, ∂u∂c, f, f′, shocks)
CHAPTER 32. OPTIMAL GROWTH II: TIME ITERATION

\[ g = \text{iterate\_updating}(\text{coleman\_single\_arg}, \text{grid}, \text{sim\_length} = 20) \]
\[ w = \text{iterate\_updating}(\text{bellman\_single\_arg}, u.(\text{grid}), \text{sim\_length} = 20) \]
\[ \text{new\_w, vf\_g} = T(w, \text{grid}, \beta, u, f, \text{shocks, compute\_policy} = \text{true}) \]

\[ \text{pf\_error} = \text{c\_star} - g \]
\[ \text{vf\_error} = \text{c\_star} - \text{vf\_g} \]

\[
\begin{align*}
\text{plot}(\text{grid}, \text{zero(\text{grid}), color = :black, lw = 1}) \\
\text{plot!}(\text{grid}, \text{pf\_error, lw = 2, alpha = 0.6, label = "policy iteration\_error"}) \\
\text{plot!}(\text{grid, vf\_error, lw = 2, alpha = 0.6, label = "value iteration\_error"}) \\
\text{plot!(legend = :bottomleft)}
\end{align*}
\]

Out[13]: compare_error (generic function with 1 method)

In [14]: compare_error(m, shocks, m.grid, m.u.(m.grid), sim_length=20)

Out[14]:

As you can see, time iteration is much more accurate for a given number of iterations.

32.6 Exercises

32.6.1 Exercise 1

Show that (9) is valid. In particular,

- Let \( v \) be strictly concave and continuously differentiable on \((0, \infty)\)
- Fix \( y \in (0, \infty) \) and show that \( MTv(y) = KMv(y) \)
32.6.2 Exercise 2

Show that $M$ is a bijection from $\mathcal{V}$ to $\mathcal{P}$.

32.6.3 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.

32.6.4 Exercise 4

Do the same exercise, but now, rather than plotting results, benchmark both approaches with 20 iterations.

32.7 Solutions

32.7.1 Solution to Exercise 1

Let $T, K, M, v$ and $y$ be as stated in the exercise.

Using the envelope theorem, one can show that $(Tv)'(y) = u'(c(y))$ where $c(y)$ solves

$$u'(c(y)) = \beta \int v'(f(y - c(y))z)f'(y - c(y))z\phi(dz)$$

Hence $MTv(y) = (u')^{-1}(u'(c(y))) = c(y)$.

On the other hand, $KMv(y)$ is the $c(y)$ that solves

$$u'(c(y)) = \beta \int (u'(Mv))(f(y - c(y))z)f'(y - c(y))z\phi(dz)$$

$$= \beta \int (u'(u'^{-1}v'))(f(y - c(y))z)f'(y - c(y))z\phi(dz)$$

$$= \beta \int u'(f(y - c(y))z)f'(y - c(y))z\phi(dz)$$

We see that $c(y)$ is the same in each case.
32.7.2 Solution to Exercise 2

We need to show that $M$ is a bijection from $\mathcal{V}$ to $\mathcal{P}$.

To see this, first observe that, in view of our assumptions above, $u'$ is a strictly decreasing continuous bijection from $(0, \infty)$ to itself.

It follows that $h$ has the same properties.

Moreover, for fixed $v \in \mathcal{V}$, the derivative $v'$ is a continuous, strictly decreasing function.

Hence, for fixed $v \in \mathcal{V}$, the map $Mv = h \circ v'$ is strictly increasing and continuous, taking values in $(0, \infty)$.

Moreover, interiority holds because $v'$ strictly dominates $u'$, implying that

$$(Mv)(y) = h(v'(y)) < h(u'(y)) = y$$

In particular, $\sigma(y) := (Mv)(y)$ is an element of $\mathcal{P}$.

To see that each $\sigma \in \mathcal{P}$ has a preimage $v \in \mathcal{V}$ with $Mv = \sigma$, fix any $\sigma \in \mathcal{P}$.

Let $v(y) := \int_0^y u'(\sigma(x))dx$ with $v(0) = 0$.

With a small amount of effort you will be able to show that $v \in \mathcal{V}$ and $Mv = \sigma$.

It’s also true that $M$ is one-to-one on $\mathcal{V}$.

To see this, suppose that $v$ and $w$ are elements of $\mathcal{V}$ satisfying $Mv = Mw$.

Then $v(0) = w(0) = 0$ and $v' = w'$ on $(0, \infty)$.

The fundamental theorem of calculus then implies that $v = w$ on $\mathbb{R}_+$.

32.7.3 Solution to Exercise 3

Here’s the code, which will execute if you’ve run all the code above.

In [15]: # Model instance with risk aversion = 1.5
   # others are the same as the previous instance
   m_ex = Model(γ = 1.5);

In [16]: function exercise2(m, shocks, g_init = m.grid, w_init = m.u.(m.grid); sim_length = 20)
   @unpack grid, β, u, ∂u∂c, f, f' = m
   # initial policy and value
   g, w = g_init, w_init
   # iteration
   bellman_single_arg(w) = T(w, grid, β, u, f, shocks)
   coleman_single_arg(g) = K(g, grid, β, ∂u∂c, f, f', shocks)
   g = iterate_updating(coleman_single_arg, grid, sim_length = 20)
   w = iterate_updating(bellman_single_arg, u.(m.grid), sim_length = 20)
   new_w, vf_g = T(w, grid, β, u, f, shocks, compute_policy = true)
   plot(grid, g, lw = 2, alpha = 0.6, label = "policy iteration")
   plot!(grid, vf_g, lw = 2, alpha = 0.6, label = "value iteration")
32.7. SOLUTIONS

```python
return plot!(legend = :topleft)
end
```

Out[16]: exercise2 (generic function with 3 methods)

In [17]: exercise2(m_ex, shocks, m.grid, m.u.(m.grid), sim_length=20)

Out[17]:

![Graph showing policy iteration and value iteration](image)

The policies are indeed close.

32.7.4 Solution to Exercise 4

Here’s the code.

It assumes that you’ve just run the code from the previous exercise

In [18]: function bellman(m, shocks)
   @unpack grid, β, u, ∂u∂c, f, f' = m
   bellman_single_arg(w) = T(w, grid, β, u, f, shocks)
   iterate_updating(bellman_single_arg, u.(grid), sim_length = 20)
end

function coleman(m, shocks)
   @unpack grid, β, ∂u∂c, f, f' = m
   coleman_single_arg(g) = K(g, grid, β, ∂u∂c, f, f', shocks)
   iterate_updating(coleman_single_arg, grid, sim_length = 20)
end

Out[18]: coleman (generic function with 1 method)
In [19]: @benchmark bellman(m_ex, shocks)

Out[19]: BenchmarkTools.Trial:
        memory estimate: 155.94 MiB
        allocs estimate: 90741
        ---------------
        minimum time: 393.432 ms (4.12% GC)
        median time: 396.649 ms (4.15% GC)
        mean time: 396.624 ms (4.45% GC)
        maximum time: 399.935 ms (5.50% GC)
        ---------------
        samples: 13
        evals/sample: 1

In [20]: @benchmark bellman(m_ex, shocks)

Out[20]: BenchmarkTools.Trial:
        memory estimate: 155.94 MiB
        allocs estimate: 90741
        ---------------
        minimum time: 390.393 ms (4.21% GC)
        median time: 394.890 ms (4.16% GC)
        mean time: 394.367 ms (4.49% GC)
        maximum time: 402.473 ms (5.44% GC)
        ---------------
        samples: 13
        evals/sample: 1
Chapter 33

Optimal Growth III: The Endogenous Grid Method

33.1 Contents

- Overview 33.2
- Key Idea 33.3
- Implementation 33.4
- Speed 33.5

33.2 Overview

We solved the stochastic optimal growth model using

1. value function iteration
2. Euler equation based time iteration

We found time iteration to be significantly more accurate at each step.

In this lecture we’ll look at an ingenious twist on the time iteration technique called the endogenous grid method (EGM).

EGM is a numerical method for implementing policy iteration invented by Chris Carroll.

It is a good example of how a clever algorithm can save a massive amount of computer time. (Massive when we multiply saved CPU cycles on each implementation times the number of implementations worldwide)

The original reference is [16].

33.3 Key Idea

Let’s start by reminding ourselves of the theory and then see how the numerics fit in.
33.3.1 Theory

Take the model set out in the time iteration lecture, following the same terminology and notation.

The Euler equation is

\[(u' \circ c^*)(y) = \beta \int (u' \circ c^*)(f(y - c^*(y))z) f'(y - c^*(y))z\phi(dz)\] (1)

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

\[u'(c) = \beta \int (u' \circ g)(f(y - c)z) f'(y - c)z\phi(dz)\] (2)

33.3.2 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 (2) 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...

33.3.3 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 \([16]\), 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
33.4 IMPLEMENTATION

\[ c_i = (u')^{-1} \left\{ \beta \int (u' \circ g)(f(k_i)z) f'(k_i) z \phi(dz) \right\} \]  \hspace{1cm} (3)

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 (2).

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.

33.4 Implementation

Let’s implement this version of the Coleman operator and see how it performs.

33.4.1 The Operator

Here’s an implementation of \(K\) using EGM as described above.

33.4.2 Setup

In [1]: using InstantiateFromURL
   # optionally add arguments to force installation: instantiate = true,[]
   # precompile = true
   github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

In [2]: using LinearAlgebra, Statistics
   using BenchmarkTools, Interpolations, Parameters, Plots, QuantEcon,[]
   #Random, Roots
   gr(fmt = :png);

In [3]: function coleman_egm(g, k_grid, \(\beta\), u', u'_inv, f, f', shocks)
   # 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'(g.(f(k) .* shocks)) .* f'(k) .* shocks
     c[i] = u'_inv(\beta * mean(vals))
   end

   # Determine endogenous grid
   y = k_grid + c  # \(y_i = k_i + c_i\)

   # Update policy function and return
   Kg = LinearInterpolation(y, c, extrapolation_bc=Line())
   Kg_f(x) = Kg(x)
   return Kg_f
end
Out[3]: coleman_egm (generic function with 1 method)

Note the lack of any root finding algorithm.

We’ll also run our original implementation, which uses an exogenous grid and requires root finding, so we can perform some comparisons

In [4]: function K!(Kg, g, grid, β, u', f, f', shocks)

    # This function requires the container of the output value as argument Kg
    # Construct linear interpolation object#
    g_func = LinearInterpolation(grid, g, extrapolation_bc = Line())

    # solve for updated consumption value#
    for (i, y) in enumerate(grid)
        function h(c)
            vals = u'.(g_func.(f(y - c) * shocks)) .* f'(y - c) .* shocks
            return u'(c) - β * mean(vals)
        end
        Kg[i] = find_zero(h, (1e-10, y - 1e-10))
    end
    return Kg
end

Out[4]: K (generic function with 1 method)

Let’s test out the code above on some example parameterizations, after the following imports.

### 33.4.3 Testing on the Log / Cobb–Douglas case

As we did for value function iteration and time iteration, let’s start by testing our method with the log-linear benchmark.

The first step is to bring in the model that we used in the Coleman policy function iteration

In [5]: # model

    Model = @with_kw (α = 0.65, # productivity parameter
                        β = 0.95, # discount factor
                        γ = 1.0, # risk aversion
                        μ = 0.0, # lognorm(μ, σ)
                        s = 0.1, # lognorm(μ, σ)
                        grid_min = 1e-6, # smallest grid point
                        grid_max = 4.0, # largest grid point
                        grid_size = 200, # grid size
                        u = γ == 1 ? log : c -> c^((1-γ)/gamma), # utility function
                        u' = c -> c^(1-γ), # u'
                        f = k -> k^α, # production function
                        f' = k -> α*k^(α-1), # f'
                        grid = range(grid_min, grid_max, length = grid_size)) # grid
Out[5]: #<NamedTuple_kw#253 (generic function with 2 methods)

Next we generate an instance

In [6]: mlog = Model(); # Log Linear model

We also need some shock draws for Monte Carlo integration

In [7]: Random.seed!(42); # For reproducible behavior.

    shock_size = 250  # Number of shock draws in Monte Carlo integral
    shocks = exp.(mlog.μ .+ mlog.s * randn(shock_size));

As a preliminary test, let’s see if \( Kc^* = c^* \), as implied by the theory

In [8]: c_star(y) = (1 - mlog.α * mlog.β) * y

    # some useful constants
    ab = mlog.α * mlog.β
    c1 = log(1 - ab) / (1 - mlog.β)
    c2 = (mlog.μ + mlog.α * log(ab)) / (1 - mlog.α)
    c3 = 1 / (1 - mlog.β)
    c4 = 1 / (1 - ab)

    v_star(y) = c1 + c2 * (c3 - c4) + c4 * log(y)

Out[8]: v_star (generic function with 1 method)

In [9]: function verify_true_policy(m, shocks, c_star)
    k_grid = m.grid
    c_star_new = coleman_egm(c_star, k_grid, m.β, m.u', m.u', m.f, m.f', shocks)
    plt = plot()
    plot!(plt, k_grid, c_star.(k_grid), lw = 2, label = "optimal policy c*")
    plot!(plt, k_grid, c_star_new.(k_grid), lw = 2, label = "Kc*")
    plot!(plt, legend = :topleft)
end

Out[9]: verify_true_policy (generic function with 1 method)

In [10]: verify_true_policy(mlog, shocks, c_star)

Out[10]: 
Notice that we’re passing \( u \) to Coleman’s EGM twice.

The reason is that, in the case of log utility, \( u'(c) = (u')^{-1}(c) = 1/c. \)

Hence \( u \) and \( u_{\text{inv}} \) are the same.

We can’t really distinguish the two plots.

In fact it’s easy to see that the difference is essentially zero:

```
In [11]: c_star_new = coleman_egm(c_star, mlog.grid, mlog.\beta, mlog.u',
                         mlog.u', mlog.f, mlog.f', shocks)

maximum(abs(c_star_new(g) - c_star(g)) for g in mlog.grid)
```

```
Out[11]: 1.332676295501878e-15
```

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: \( c(y) = y \)

```
In [12]: n = 15

function check_convergence(m, shocks, c_star, g_init, n_iter)
    k_grid = m.grid
    g = g_init
    plt = plot()
    plot!(plt, m.grid, g.(m.grid),
    color = RGBA(0,0,0,1), lw = 2, alpha = 0.6, label = "initial")
    condition c(y) =
    for i in 1:n_iter
        new_g = coleman_egm(g, k_grid, m.\beta, m.u', m.u', m.f, m.f', shocks)
        g = new_g
        plot!(plt, k_grid, new_g.(k_grid), alpha = 0.6, color = RGBA(0,0,0,1/i/n_iter),
```
33.5. SPEED

1), lw = 2, label = "")

end

plot!(plt, k_grid, c_star.(k_grid),
    color = :black, lw = 2, alpha = 0.8, label = "true policy"
    function c"

) plot!(plt, legend = :topleft)
end

Out[12]: check_convergence (generic function with 1 method)

In [13]: check_convergence(mlog, shocks, c_star, identity, n)

Out[13]:

![Graph showing convergence of policy]

We see that the policy has converged nicely, in only a few steps.

33.5 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

In [14]: mcrra = Model(α = 0.65, β = 0.95, γ = 1.5)
    u'_inv(c) = c^(-1 / mcrra.γ)
Out[14]: `u'_inv (generic function with 1 method)

Here's the result

In [15]:
crra_coleman(g, m, shocks) = K(g, m.grid, m.β, m.u', m.f, m.f', shocks)
crra_coleman_egm(g, m, shocks) = coleman_egm(g, m.grid, m.β, m.u',
                                    u'_inv, m.f, m.f', shocks)

function coleman(m = m, shocks = shocks; sim_length = 20)
g = m.grid
    for i in 1:sim_length
        g = crra_coleman(g, m, shocks)
    end
end

function egm(m, g = identity, shocks = shocks; sim_length = 20)
    for i in 1:sim_length
        g = crra_coleman_egm(g, m, shocks)
    end
end

Out[15]: `egm (generic function with 3 methods)

In [16]: `@benchmark coleman($mcrra)

Out[16]: BenchmarkTools.Trial:
    memory estimate: 1.03 GiB
    allocs estimate: 615012
                  ---------------
    minimum time:  8.433 s (1.38% GC)
    median time:   8.433 s (1.38% GC)
    mean time:     8.433 s (1.38% GC)
    maximum time:  8.433 s (1.38% GC)
                  ---------------
    samples:       1
    evals/sample:  1

In [17]: `@benchmark egm($mcrra)

Out[17]: BenchmarkTools.Trial:
    memory estimate: 18.50 MiB
    allocs estimate: 76226
                  ---------------
    minimum time:  180.788 ms (0.00% GC)
    median time:   183.655 ms (0.00% GC)
    mean time:     184.908 ms (1.14% GC)
    maximum time:  190.417 ms (3.11% GC)
                  ---------------
    samples:       28
    evals/sample:  1

We see that the EGM version is about 30 times faster.

At the same time, the absence of numerical root finding means that it is typically more accurate at each step as well.
Chapter 34

LQ Dynamic Programming
Problems

34.1 Contents

- Overview 34.2
- Introduction 34.3
- Optimality – Finite Horizon 34.4
- Implementation 34.5
- Extensions and Comments 34.6
- Further Applications 34.7
- Exercises 34.8
- Solutions 34.9

34.2 Overview

Linear quadratic (LQ) control refers to a class of dynamic optimization problems that have found applications in almost every scientific field.

This lecture provides an introduction to LQ control and its economic applications.

As we will see, LQ systems have a simple structure that makes them an excellent workhorse for a wide variety of economic problems.

Moreover, while the linear-quadratic structure is restrictive, it is in fact far more flexible than it may appear initially.

These themes appear repeatedly below.

Mathematically, LQ control problems are closely related to the Kalman filter.

- Recursive formulations of linear-quadratic control problems and Kalman filtering problems both involve matrix Riccati equations.
- Classical formulations of linear control and linear filtering problems make use of similar matrix decompositions (see for example this lecture and this lecture).

In reading what follows, it will be useful to have some familiarity with

- matrix manipulations
- vectors of random variables

609
CHAPTER 34. LQ DYNAMIC PROGRAMMING PROBLEMS

- dynamic programming and the Bellman equation (see for example this lecture and this lecture)

For additional reading on LQ control, see, for example,

- [68], chapter 5
- [39], chapter 4
- [53], section 3.5

In order to focus on computation, we leave longer proofs to these sources (while trying to pro-
vide as much intuition as possible).

34.3 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 opti-
mization problem.

34.3.1 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

\[
x_{t+1} = Ax_t + Bu_t + Cw_{t+1}, \quad t = 0, 1, 2, \ldots
\]  

(1)

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 stan-
dard normal, we can write the system as
34.3. INTRODUCTION

\[ a_{t+1} = (1 + r)a_t - c_t + \sigma w_{t+1} \]

This is clearly a special case of (1), with assets being the state and consumption being the control.

**Example 2**

One unrealistic feature of the previous model is that non-financial income has a zero mean and is often negative.

This can easily be overcome by adding a sufficiently large mean.

Hence in this example we take \( y_t = \sigma w_{t+1} + \mu \) for some positive real number \( \mu \).

Another alteration that’s useful to introduce (we’ll see why soon) is to change the control variable from consumption to the deviation of consumption from some “ideal” quantity \( \bar{c} \).

(Most parameterizations will be such that \( \bar{c} \) is large relative to the amount of consumption that is attainable in each period, and hence the household wants to increase consumption)

For this reason, we now take our control to be \( u_t := c_t - \bar{c} \).

In terms of these variables, the budget constraint \( a_{t+1} = (1 + r)a_t - c_t + y_t \) becomes

\[ a_{t+1} = (1 + r)a_t - u_t - \bar{c} + \sigma w_{t+1} + \mu \quad (2) \]

How can we write this new system in the form of equation (1)?

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

\[
\begin{pmatrix}
    a_{t+1} \\
    1
\end{pmatrix} =
\begin{pmatrix}
    1 + r & -\bar{c} + \mu \\
    0 & 1
\end{pmatrix}
\begin{pmatrix}
    a_t \\
    1
\end{pmatrix} +
\begin{pmatrix}
    -1 \\
    0
\end{pmatrix} u_t +
\begin{pmatrix}
    \sigma \\
    0
\end{pmatrix} w_{t+1}
\]

(3)

then the first row is equivalent to (2).

Moreover, the model is now linear, and can be written in the form of (1) by setting

\[
x_t :=
\begin{pmatrix}
    a_t \\
    1
\end{pmatrix}, \quad A :=
\begin{pmatrix}
    1 + r & -\bar{c} + \mu \\
    0 & 1
\end{pmatrix}, \quad B :=
\begin{pmatrix}
    -1 \\
    0
\end{pmatrix}, \quad C :=
\begin{pmatrix}
    \sigma \\
    0
\end{pmatrix}
\]

(4)

In effect, we’ve bought ourselves linearity by adding another state.

**34.3.2 Preferences**

In the LQ model, the aim is to minimize a flow of losses, where time-\( t \) loss is given by the quadratic expression
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 [39] 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

$$x_t' R x_t + u_t' Q u_t$$

Thus, for both the state and the control, loss is measured as squared distance from the origin.

(In fact the general case (5) 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 $\bar{c}$.

### 34.4 Optimality – Finite Horizon

Let’s now be precise about the optimization problem we wish to consider, and look at how to solve it.

#### 34.4.1 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

\[
\mathbb{E} \left\{ \sum_{t=0}^{T-1} \beta^t (x'_t R x_t + u'_t Q u_t) + \beta^T x'_T R_f x_T \right\}
\]  \hspace{1cm} (6)

subject to the law of motion (1) 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 \( w_1, \ldots, w_T \)

### 34.4.2 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 \( x_t \).
34.4.3 Solution

To solve the finite horizon LQ problem we can use a dynamic programming strategy based on backwards induction that is conceptually similar to the approach adopted in this lecture.

For reasons that will soon become clear, we first introduce the notation $J_T(x) = x' R_f x$.

Now consider the problem of the decision maker in the second to last period.

In particular, let the time be $T - 1$, and suppose that the state is $x_{T-1}$.

The decision maker must trade off current and (discounted) final losses, and hence solves

$$\min_u \{ x_{T-1}' R x_{T-1} + u' Q u + \beta \mathbb{E} J_T(A x_{T-1} + B u + C w_T) \}$$

At this stage, it is convenient to define the function

$$J_{T-1}(x) = \min_u \{ x' R x + u' Q u + \beta \mathbb{E} J_T(A x + B u + C w_T) \}$$

(7)

The function $J_{T-1}(x)$ will be called the $T-1$ value function, and $J_{T-1}(x)$ can be thought of as representing total “loss-to-go” from state $x$ at time $T - 1$ when the decision maker behaves optimally.

Now let’s step back to $T - 2$.

For a decision maker at $T - 2$, the value $J_{T-1}(x)$ plays a role analogous to that played by the terminal loss $J_T(x) = x' R_f x$ for the decision maker at $T - 1$.

That is, $J_{T-1}(x)$ summarizes the future loss associated with moving to state $x$.

The decision maker chooses her control $u$ to trade off current loss against future loss, where

- the next period state is $x_{T-1} = A x_{T-2} + B u + C w_{T-1}$, and hence depends on the choice of current control
- the “cost” of landing in state $x_{T-1}$ is $J_{T-1}(x_{T-1})$

Her problem is therefore

$$\min_u \{ x_{T-2}' R x_{T-2} + u' Q u + \beta \mathbb{E} J_{T-1}(A x_{T-2} + B u + C w_{T-1}) \}$$

Letting

$$J_{T-2}(x) = \min_u \{ x' R x + u' Q u + \beta \mathbb{E} J_{T-1}(A x + B u + C w_{T-1}) \}$$

the pattern for backwards induction is now clear.

In particular, we define a sequence of value functions $\{ J_0, \ldots, J_T \}$ via

$$J_{t-1}(x) = \min_u \{ x' R x + u' Q u + \beta \mathbb{E} J_t(A x + B u + C w_t) \} \quad \text{and} \quad J_T(x) = x' R_f x$$

The first equality is the Bellman equation from dynamic programming theory specialized to the finite horizon LQ problem.

Now that we have $\{ J_0, \ldots, J_T \}$, we can obtain the optimal controls.
As a first step, let’s find out what the value functions look like.

It turns out that every $J_t$ has the form $J_t(x) = x'P_t x + d_t$ where $P_t$ is a $n \times n$ matrix and $d_t$ is a constant.

We can show this by induction, starting from $P_T := R_f$ and $d_T = 0$.

Using this notation, (7) becomes

$$J_{T-1}(x) = \min_u \{x'R + u'Q + \beta E(Ax + Bu + Cw_T)'P_T(Ax + Bu + Cw_T)\}$$  \hspace{1cm} (8)

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 \textit{matrix calculus}, this gives

$$u = -(Q + \beta B'P_T B)^{-1}\beta B'P_T Ax$$ \hspace{1cm} (9)

Plugging this back into (8) and rearranging yields

$$J_{T-1}(x) = x'P_{T-1} x + d_{T-1}$$

where

$$P_{T-1} = R - \beta^2 A'P_T B(Q + \beta B'P_T B)^{-1}B'P_T A + \beta A'P_T A$$ \hspace{1cm} (10)

and

$$d_{T-1} := \beta \text{trace}(C'P_T C)$$ \hspace{1cm} (11)

(The algebra is a good exercise — we’ll leave it up to you).

If we continue working backwards in this manner, it soon becomes clear that $J_t(x) = x'P_t x + d_t$ as claimed, where $\{P_t\}$ and $\{d_t\}$ satisfy the recursions

$$P_{t-1} = R - \beta^2 A'P_t B(Q + \beta B'P_t B)^{-1}B'P_t A + \beta A'P_t A \quad \text{with} \quad P_T = R_f$$ \hspace{1cm} (12)

and

$$d_{t-1} = \beta(d_t + \text{trace}(C'P_t C)) \quad \text{with} \quad d_T = 0$$ \hspace{1cm} (13)

Recalling (9), the minimizers from these backward steps are

$$u_t = -F_t x_t \quad \text{where} \quad F_t := (Q + \beta B'P_{t+1}B)^{-1}\beta B'P_{t+1} A$$ \hspace{1cm} (14)

These are the linear optimal control policies we discussed above.

In particular, the sequence of controls given by (14) and (1) solves our finite horizon LQ problem.

Rephrasing this more precisely, the sequence $u_0, \ldots, u_{T-1}$ given by
\[ u_t = -F_t x_t \quad \text{with} \quad x_{t+1} = (A - BF_t)x_t + C w_{t+1} \]  

for \( t = 0, \ldots, T - 1 \) attains the minimum of (6) subject to our constraints.

### 34.5 Implementation

We will use code from \texttt{lqcontrol.jl} in \texttt{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 \texttt{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 \texttt{None} in the infinite horizon case
    * set \( C = \texttt{None} \) (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:**
  - \texttt{update_values} — shifts \( d_t, P_t, F_t \) to their \( t - 1 \) values via (12), (13) and (14)
  - \texttt{stationary_values} — computes \( P, d, F \) in the infinite horizon case
  - \texttt{compute_sequence} — simulates the dynamics of \( x_t, u_t, w_t \) given \( x_0 \) and assuming standard normal shocks

### 34.5.1 An Application

Early Keynesian models assumed that households have a constant marginal propensity to consume from current income.

Data contradicted the constancy of the marginal propensity to consume.

In response, Milton Friedman, Franco Modigliani and others built models based on a consumer’s preference for an intertemporally smooth consumption stream.

(See, for example, [32] or [79]).

One property of those models is that households purchase and sell financial assets to make consumption streams smoother than income streams.

The household savings problem outlined above captures these ideas.

The optimization problem for the household is to choose a consumption sequence in order to minimize

\[ \mathbb{E} \left\{ \sum_{t=0}^{T-1} \beta^t (c_t - \bar{c})^2 + \beta^T q a_T^2 \right\} \]  

subject to the sequence of budget constraints \( a_{t+1} = (1 + r)a_t - c_t + y_t, \ t \geq 0. \)

Here \( q \) is a large positive constant, the role of which is to induce the consumer to target zero debt at the end of her life.
(Without such a constraint, the optimal choice is to choose \( c_t = \bar{c} \) in each period, letting assets adjust accordingly).

As before we set \( y_t = \sigma w_{t+1} + \mu \) and \( u_t := c_t - \bar{c} \), after which the constraint can be written as in (2).

We saw how this constraint could be manipulated into the LQ formulation \( x_{t+1} = Ax_t + Bu_t + Cw_{t+1} \) by setting \( x_t = (a_t, 1)' \) and using the definitions in (4).

To match with this state and control, the objective function (16) can be written in the form of (6) 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 (12) and (14).

After generating shocks \( w_1, ..., w_T \), the dynamics for assets and consumption can be simulated via (15).

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.

### 34.5.2 Setup

In [1]: using InstantiateFromURL
   # optionally add arguments to force installation: instantiate = true, precompile = true
   github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

In [2]: using LinearAlgebra, Statistics, Plots, Plots.PlotMeasures, QuantEcon
   gr(fmt = :png);

In [3]: # model parameters
   \r = 0.05
   \beta = 1 / (1 + \r)
   \T = 45
   \bar{c} = 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 + \r - \bar{c} + \mu; 0 1]
   B = [-1.0, 0]
   C = [\sigma, 0]

   # compute solutions and simulate
\[
lq = \text{QuantEcon.LQ}(Q, R, A, B, C; \text{bet} = \beta, \text{capT} = T, \text{rf} = Rf)
\]
\[
x0 = [0.0, 1]
\]
\[
\text{xp, up, wp = compute_sequence(lq, x0)}
\]

# convert back to assets, consumption and income
\[
\text{assets} = \text{vec(xp[1, :])} \ # a_t
\]
\[
\text{c} = \text{vec(up} + c) \ # c_t
\]
\[
\text{income} = \text{vec}(\sigma \ast \text{wp[1, 2:end]} .+ \mu) \ # y_t
\]

# plot results
\[
\text{p = plot([assets, c, zeros(T + 1), income, cumsum(income .- \mu)]),}
\]
\[
\text{lab = ["assets" "consumption" "non-financial income" "cumulative unanticipated income"],}
\]
\[
\text{color = ["blue" "green" "black" "orange" "red"],}
\]
\[
\text{xaxis = "Time", layout = (2, 1),}
\]
\[
\text{bottom_margin = 20mm, size = (600, 600))}
\]

Out[3]:

The top panel shows the time path of consumption \(c_t\) and income \(y_t\) in the simulation.
As anticipated by the discussion on consumption smoothing, the time path of consumption is much smoother than that for income.

(But note that consumption becomes more irregular towards the end of life, when the zero final asset requirement impinges more on consumption choices).

The second panel in the figure shows that the time path of assets $a_t$ is closely correlated with cumulative unanticipated income, where the latter is defined as

$$z_t := \sum_{j=0}^{t} \sigma w_t$$

A key message is that unanticipated windfall gains are saved rather than consumed, while unanticipated negative shocks are met by reducing assets.

(Again, this relationship breaks down towards the end of life due to the zero final asset requirement).

These results are relatively robust to changes in parameters.

For example, let’s increase $\beta$ from $1/(1 + r) \approx 0.952$ to $0.96$ while keeping other parameters fixed.

This consumer is slightly more patient than the last one, and hence puts relatively more weight on later consumption values.

```
In [4]: # compute solutions and simulate
lq = QuantEcon.LQ(Q, R, A, B, C; bet = 0.96, capT = T, rf = Rf)
x0 = [0.0, 1]
xp, up, wp = compute_sequence(lq, x0)

# convert back to assets, consumption and income
assets = vec(xp[1, :]) # a_t
income = vec(sigma * wp[1, 2:end] + mu) # y_t

# plot results
p = plot([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.

### 34.6 Extensions and Comments

Let’s now consider a number of standard extensions to the LQ problem treated above.

#### 34.6.1 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 [40], section 2.4.

### 34.6.2 Adding a Cross-Product Term

In some LQ problems, preferences include a cross-product term $u_t' N x_t$, so that the objective function becomes

$$
E \left\{ \sum_{t=0}^{T-1} \beta^t (x_t' R x_t + u_t' Q u_t + 2u_t' N x_t) + \beta^T x_T' R_T x_T \right\}
$$

Our results extend to this case in a straightforward way.

The sequence $\{P_t\}$ from (12) becomes

$$
P_{t-1} = R - (\beta B' P_t A + N)' (Q + \beta B' P_t B)^{-1} (\beta B' P_t A + N) + \beta A' P_t A \quad \text{with} \quad P_T = R_f \quad (18)
$$

The policies in (14) are modified to

$$
u_t = -F_t x_t \quad \text{where} \quad F_t := (Q + \beta B' P_{t+1} B)^{-1} (\beta B' P_{t+1} A + N) \quad (19)
$$

The sequence $\{d_t\}$ is unchanged from (13).

We leave interested readers to confirm these results (the calculations are long but not overly difficult).

### 34.6.3 Infinite Horizon

Finally, we consider the infinite horizon case, with cross-product term, unchanged dynamics and objective function given by

$$
E \left\{ \sum_{t=0}^{\infty} \beta^t (x_t' R x_t + u_t' Q u_t + 2u_t' N x_t) \right\}
$$

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.

$$
P = R - (\beta B' P A + N)' (Q + \beta B' P B)^{-1} (\beta B' P A + N) + \beta A' P A \quad (21)
$$
Equation (21) is also called the \textit{LQ Bellman equation}, and the map that sends a given \( P \) into the right-hand side of (21) is called the \textit{LQ Bellman operator}.

The stationary optimal policy for this model is

\begin{equation}
\begin{aligned}
u &= -Fx \\
F &= (Q + \beta B'PB)^{-1}(\beta B'PA + N)
\end{aligned}
\end{equation}

The sequence \( \{d_t\} \) from (13) is replaced by the constant value

\begin{equation}
d := \text{trace}(C'PC) \frac{\beta}{1 - \beta}
\end{equation}

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.

\section{34.6.4 Certainty Equivalence}

Linear quadratic control problems of the class discussed above have the property of \textit{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 (22) or (19).

It follows that we can ignore uncertainty when solving for optimal behavior, and plug it back in when examining optimal state dynamics.

\section{34.7 Further Applications}

\subsection{34.7.1 Application 1: Age-Dependent Income Process}

Previously we studied a permanent income model that generated consumption smoothing.

One unrealistic feature of that model is the assumption that the mean of the random income process does not depend on the consumer’s age.

A more realistic income profile is one that rises in early working life, peaks towards the middle and maybe declines toward end of working life, and falls more during retirement.

In this section, we will model this rise and fall as a symmetric inverted “U” using a polynomial in age.

As before, the consumer seeks to minimize

\begin{equation}
\mathbb{E} \left\{ \sum_{t=0}^{T-1} \beta^t (c_t - \bar{c})^2 + \beta^T qa_T^2 \right\}
\end{equation}

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

$$a_{t+1} = (1 + r)a_t - u_t - \bar{c} + m_1 t + m_2 t^2 + \sigma w_{t+1}$$  \hspace{1cm} (25)$$

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

$$x_t := \begin{pmatrix} a_t \\ 1 \\ t \\ t^2 \end{pmatrix}, \quad A := \begin{pmatrix} 1 + r & -\bar{c} & m_1 & m_2 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 2 & 1 \end{pmatrix}, \quad B := \begin{pmatrix} -1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \quad C := \begin{pmatrix} \sigma \\ 0 \\ 0 \end{pmatrix}$$  \hspace{1cm} (26)$$

If you expand the expression $x_{t+1} = Ax_t + Bu_t + Cw_{t+1}$ using this specification, you will find that assets follow (25) as desired, and that the other state variables also update appropriately.

To implement preference specification (24) we take

$$Q := 1, \quad R := \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad \text{ and } \quad R_f := \begin{pmatrix} q & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$  \hspace{1cm} (27)$$

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.

### 34.7.2 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

$$
y_t = \begin{cases} 
p(t) + \sigma w_{t+1} & \text{if } t \leq K \\
\frac{s}{s} & \text{otherwise}
\end{cases}
$$

(28)

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 (16).

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 (28) at $K$ makes it very difficult to express the law of motion as a fixed-coefficient linear system.

However, we can still use our LQ methods here by suitably linking two component LQ problems.

These two LQ problems describe the consumer’s behavior during her working life (lq_working) and retirement (lq_retired).

(This is possible because in the two separate periods of life, the respective income processes [polynomial trend and constant] each fit the LQ framework)

The basic idea is that although the whole problem is not a single time-invariant LQ problem, it is still a dynamic programming problem, and hence we can use appropriate Bellman equations at every stage.

Based on this logic, we can

1. solve lq_retired by the usual backwards induction procedure, iterating back to the start of retirement

2. take the start-of-retirement value function generated by this process, and use it as the terminal condition $R_f$ to feed into the lq_working specification

3. solve lq_working by backwards induction from this choice of $R_f$, iterating back to the start of working life

This process gives the entire life-time sequence of value functions and optimal policies.
CHAPTER 34. LQ DYNAMIC PROGRAMMING PROBLEMS

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.

34.7.3 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
34.7. FURTHER APPLICATIONS

\[
\mathbb{E}\left\{ \sum_{t=0}^{\infty} \beta^t \pi_t \right\} \quad \text{where} \quad \pi_t := p_t q_t - cq_t - \gamma(q_{t+1} - q_t)^2
\]  

(29)

Here

- \( \gamma(q_{t+1} - q_t)^2 \) represents adjustment costs
- \( c \) is average cost of production

This can be formulated as an LQ problem and then solved and simulated, but first let’s study the problem and try to get some intuition.

One way to start thinking about the problem is to consider what would happen if \( \gamma = 0 \).

Without adjustment costs there is no intertemporal trade-off, so the monopolist will choose output to maximize current profit in each period.

It’s not difficult to show that profit-maximizing output is

\[
\bar{q}_t := \frac{a_0 - c + d_t}{2a_1}
\]

In light of this discussion, what we might expect for general \( \gamma \) is that

- if \( \gamma \) is close to zero, then \( q_t \) will track the time path of \( \bar{q}_t \) relatively closely
- if \( \gamma \) is larger, then \( q_t \) will be smoother than \( \bar{q}_t \), as the monopolist seeks to avoid adjustment costs

This intuition turns out to be correct.

The following figures show simulations produced by solving the corresponding LQ problem.

The only difference in parameters across the figures is the size of \( \gamma \).
To produce these figures we converted the monopolist problem into an LQ problem. The key to this conversion is to choose the right state — which can be a bit of an art. Here we take $x_t = (\tilde{q}_t \ q_t \ 1)'$, while the control is chosen as $u_t = q_{t+1} - q_t$.

We also manipulated the profit function slightly. In (29), current profits are $\pi_t := p_t q_t - cq_t - \gamma (q_{t+1} - q_t)^2$. Let’s now replace $\pi_t$ in (29) with $\hat{\pi}_t := \pi_t - a_t \tilde{q}_t^2$. 
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 (29), and optimizers are not affected by constant terms)

The reason for making this substitution is that, as you will be able to verify, \( \tilde{\pi}_t \) reduces to the simple quadratic

\[
\tilde{\pi}_t = -a_1(q_t - \bar{q}_t)^2 - \gamma u_t^2
\]

After negation to convert to a minimization problem, the objective becomes

\[
\min \mathbb{E} \sum_{t=0}^{\infty} \beta^t \left\{ a_1(q_t - \bar{q}_t)^2 + \gamma u_t^2 \right\}
\]

It’s now relatively straightforward to find \( R \) and \( Q \) such that (30) can be written as (20).

Furthermore, the matrices \( A, B \) and \( C \) from (1) 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.

### 34.8 Exercises

#### 34.8.1 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 \( q = 10^4 \).

#### 34.8.2 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 \texttt{lq\_working}, \( x_t, A, B, C \) can be chosen as in (26).

- Recall that \( m_1, m_2 \) are chosen so that \( p(K) = \mu \) and \( p(2K) = 0 \).

For \texttt{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 \texttt{lq\_retired}, set preferences as in (27).

For \texttt{lq\_working}, preferences are the same, except that \( R_f \) should be replaced by the final value function that emerges from iterating \texttt{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.

### 34.8.3 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).

### 34.9 Solutions

#### 34.9.1 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
- $p(T) = 0$.

In [5]:

```python
# model parameters
r = 0.05
beta = 1 / (1 + r)
T = 50
c = 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 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 = QuantEcon.LQ(Q, R, A, B, C; bet=beta, capT=T, rf=Rf)
x0 = [0.0, 1, 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) # 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)],
          lab = ["non-financial income" "assets" "consumption" ""],
          color = ["orange" "blue" "green" "black"],
          xaxis = "Time", layout = (2,1),
          bottom_margin = 20mm, size = (600, 600))

Out[5]:

34.9.2 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 pro-
gramming problems as described in the lecture.

In [6]: # model parameters
    r = 0.05
    \beta = 1/(1 + r)
    T = 60
    K = 40
    c = 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 \ 0 \ 0; 
         0 \ 1 \ 0 \ 0; 
         0 \ 1 \ 1 \ 0; 
         0 \ 1 \ 2 \ 1]
    B = [-1.0, 0, 0, 0]
    C = zeros(4)

    # initialize LQ instance for retired agent
    lq_retired = QuantEcon.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
    lq_retired_proxy = QuantEcon.LQ(Q, R, A, B, C; bet = \beta, capT = T - K, rf = Rf)

    # 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 - \frac{c}{K} m1 m2; 
         0 \ 1 \ 0 \ 0; 
         0 \ 1 \ 1 \ 0; 
         0 \ 1 \ 2 \ 1]
    B = [-1.0, 0, 0, 0]
    C = [\sigma, 0, 0, 0]

    # set up working life LQ instance with terminal Rf from lq_retired
    lq_working = QuantEcon.LQ(Q, R, A, B, C; bet = \beta, capT = K, rf = Rf2)

    # simulate working state / control paths
34.9. SOLUTIONS

\[ x_0 = [0, 1, 0, 0] \]
\[ \text{xp}_w, \text{up}_w, \text{wp}_w = \text{compute_sequence}(\text{lq}_\text{working}, x_0) \]

# simulate retirement paths (note the initial condition)
\[ \text{xp}_r, \text{up}_r, \text{wp}_r = \text{compute_sequence}(\text{lq}_\text{retired}, \text{xp}_w[: \text{end}]) \]

# convert results back to assets, consumption and income
\[ \text{xp} = [\text{xp}_w \text{xp}_r[: \text{end}]] \]
\[ \text{assets} = \text{vec}(\text{xp}[1, :]) \] # assets
\[ \text{up} = [\text{up}_w \text{up}_r] \]
\[ \text{c} = \text{vec}(\text{up} + \cdot) \] # consumption

\[ \text{time} = 1:K \]
\[ \text{income}_w = \sigma \cdot \text{vec}(\text{wp}_w[1, 2:K+1]) + m_1 \cdot \text{time} + m_2 \cdot \text{time}^2 \] # income
\[ \text{income}_r = \text{ones}(T - K) \cdot s \]
\[ \text{income} = [\text{income}_w; \text{income}_r] \]

# plot results
\[ \text{p2} = \text{plot}([\text{income}, \text{assets}, \text{c}, \text{zeros}(T + 1)], \]
\[ \quad \text{lab} = ["non-financial income" "assets" "consumption" ""], \]
\[ \quad \text{color} = [\text{orange} \ : \text{blue} \ : \text{green} \ : \text{black}], \]
\[ \quad \text{xaxis} = "Time", \text{layout} = (2, 1), \]
\[ \quad \text{bottom_margin} = 20\text{mm}, \text{size} = (600, 600)) \]

Out[6]:
34.9.3 Exercise 3

The first task is to find the matrices $A, B, C, Q, R$ that define the LQ problem.

Recall that $x_t = (\bar{q}_t, q_t, 1)'$, while $u_t = q_{t+1} - q_t$.

Letting $m_0 := (a_0 - c)/2a_1$ and $m_1 := 1/2a_1$, we can write $\bar{q}_t = m_0 + m_1d_t$, and then, with some manipulation

$$\bar{q}_{t+1} = m_0(1 - \rho) + \rho\bar{q}_t + m_1\sigma w_{t+1}$$

By our definition of $u_t$, the dynamics of $q_t$ are $q_{t+1} = q_t + u_t$.

Using these facts you should be able to build the correct $A, B, C$ matrices (and then check them against those found in the solution code below).

Suitable $R, Q$ matrices can be found by inspecting the objective function, which we repeat here for convenience:

$$\min E \left\{ \sum_{t=0}^{\infty} \beta^t a_1(q_t - \bar{q}_t)^2 + \gamma u_t^2 \right\}$$
Our solution code is

```
In [7]: # 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]
    C = [m1 * σ, 0, 0]

    lq = QuantEcon.LQ(Q, R, A, B, C; bet = β)

# simulate state / control paths
    xθ = [m0, 2, 1]
    xp, up, wp = compute_sequence(lq, xθ, 150)
    q̇ = vec(xp[1, :])
    q = vec(xp[2, :])

# plot simulation results
    p3 = plot(eachindex(q), [q̇ q],
              lab = ["q bar" "q"],
              color = [:black :blue],
              xaxis = "Time", title = "Dynamics with γ = $γ",
              bottom_margin = 20*mm, top_margin = 10*mm,
              size = (700, 500))
```

Out[7]:
Dynamics with $\gamma = 1.0$
Chapter 35

Optimal Savings I: The Permanent Income Model

35.1 Contents

- Overview 35.2
- The Savings Problem 35.3
- Alternative Representations 35.4
- Two Classic Examples 35.5
- Further Reading 35.6
- Appendix: the Euler Equation 35.7

35.2 Overview

This lecture describes a rational expectations version of the famous permanent income model of Milton Friedman [32].

Robert Hall cast Friedman’s model within a linear-quadratic setting [36].

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 Hall’s [36] and chapter 2 of [68].

35.2.1 Setup

In [1]: using InstantiateFromURL
   # optionally add arguments to force installation: instantiate = true,
   # precompile = true
   github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")
In this section we state and solve the savings and consumption problem faced by the consumer.

35.3.1 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).

35.3.2 The Decision Problem

A consumer has preferences over consumption streams that are ordered by the utility functional

\[
\mathbb{E}_0 \left[ \sum_{t=0}^{\infty} \beta^t u(c_t) \right]
\]

where
35.3. THE SAVINGS PROBLEM

- $\mathbb{E}_t$ is the mathematical expectation conditioned on the consumer’s time $t$ information
- $c_t$ is time $t$ consumption
- $u$ is a strictly concave one-period utility function
- $\beta \in (0, 1)$ is a discount factor

The consumer maximizes (1) by choosing a consumption, borrowing plan $\{c_t, b_{t+1}\}_{t=0}^{\infty}$ subject to the sequence of budget constraints

$$c_t + b_t = \frac{1}{1 + r} b_{t+1} + y_t \quad t \geq 0$$

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.

35.3.3 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.

$$z_{t+1} = A z_t + C w_{t+1}$$
$$y_t = U z_t$$

where

- $\{w_t\}$ is an iid vector process with $\mathbb{E} w_t = 0$ and $\mathbb{E} w_t w'_t = I$
- the spectral radius of $A$ satisfies $\rho(A) < \sqrt{1/\beta}$
- $U$ is a selection vector that pins down $y_t$ as a particular linear combination of components of $z_t$.

The restriction on $\rho(A)$ prevents income from growing so fast that discounted geometric sums of some quadratic forms to be described below become infinite.

Regarding preferences, we assume the quadratic utility function

$$u(c_t) = -(c_t - \gamma)^2$$

where $\gamma$ is a bliss level of consumption

Note

Along with this quadratic utility specification, we allow consumption to be negative. However, by choosing parameters appropriately, we can make the probability that the model generates negative consumption paths over finite time horizons as low as desired.

Finally, we impose the no Ponzi scheme condition
This condition rules out an always-borrow scheme that would allow the consumer to enjoy bliss consumption forever.

### 35.3.4 First-Order Conditions

First-order conditions for maximizing (1) subject to (2) are

$$
\mathbb{E}_t[u'(c_{t+1})] = u'(c_t), \quad t = 0, 1, \ldots
$$

These optimality conditions are also known as Euler equations.

If you’re not sure where they come from, you can find a proof sketch in the appendix.

With our quadratic preference specification, (5) has the striking implication that consumption follows a martingale:

$$
\mathbb{E}_t[c_{t+1}] = c_t
$$

(In fact quadratic preferences are necessary for this conclusion Section ??)

One way to interpret (6) is that consumption will change only when “new information” about permanent income is revealed.

These ideas will be clarified below.

### 35.3.5 The Optimal Decision Rule

Now let’s deduce the optimal decision rule Section ??.

**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 (6)

2. the period-by-period budget constraint (2), and

3. the boundary condition (4)

To accomplish this, observe first that (4) implies \(\lim_{t \to \infty} \beta^t b_{t+1} = 0\).

Using this restriction on the debt path and solving (2) forward yields
\[ b_t = \sum_{j=0}^{\infty} \beta^j (y_{t+j} - c_{t+j}) \] (7)

Take conditional expectations on both sides of (7) and use the martingale property of consumption and the law of iterated expectations to deduce

\[ b_t = \sum_{j=0}^{\infty} \beta^j \mathbb{E}_t[y_{t+j}] - \frac{c_t}{1 - \beta} \] (8)

Expressed in terms of \( c_t \) we get

\[ c_t = (1 - \beta) \left[ \sum_{j=0}^{\infty} \beta^j \mathbb{E}_t[y_{t+j}] - b_t \right] = \frac{r}{1+r} \left[ \sum_{j=0}^{\infty} \beta^j \mathbb{E}_t[y_{t+j}] - b_t \right] \] (9)

where the last equality uses \((1 + r)\beta = 1\).

These last two equations assert that consumption equals economic income

- **financial wealth** equals \(-b_t\)
- **non-financial wealth** equals \(\sum_{j=0}^{\infty} \beta^j \mathbb{E}_t[y_{t+j}]\)
- **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 \([b_t, z_t]\).

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 consumer’s 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

\[ \sum_{j=0}^{\infty} \beta^j \mathbb{E}_t[y_{t+j}] = \mathbb{E}_t \left[ \sum_{j=0}^{\infty} \beta^j y_{t+j} \right] = U(I - \beta A)^{-1}z_t \]

Combining this with (9) gives

\[ c_t = \frac{r}{1+r} [U(I - \beta A)^{-1}z_t - b_t] \] (10)
Using this equality to eliminate $c_t$ in the budget constraint (2) gives

$$b_{t+1} = (1 + r)(b_t + c_t - y_t)$$
$$= (1 + r)b_t + r[U(I - βA)^{-1}z_t - b_t] - (1 + r)Uz_t$$
$$= b_t + U[r(I - βA)^{-1} - (1 + r)I]z_t$$
$$= b_t + U(I - βA)^{-1}(A - I)z_t$$

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 $(1 + r)β = 1$ and $(I - βA)^{-1} = \sum_{j=0}^{∞} β^j A^j$.

We’ve now successfully written $c_t$ and $b_{t+1}$ as functions of $b_t$ and $z_t$.

**A State-Space Representation**

We can summarize our dynamics in the form of a linear state-space system governing consumption, debt and income:

$$z_{t+1} = A z_t + C w_{t+1}$$
$$b_{t+1} = b_t + U[(I - βA)^{-1}(A - I)]z_t$$
$$y_t = U z_t$$
$$c_t = (1 - β)[U(I - βA)^{-1}z_t - b_t]$$

To write this more succinctly, let

$$x_t = \begin{bmatrix} z_t \\ b_t \end{bmatrix}, \quad \tilde{A} = \begin{bmatrix} A & 0 \\ U(I - βA)^{-1}(A - I) & 1 \end{bmatrix}, \quad \tilde{C} = \begin{bmatrix} C \\ 0 \end{bmatrix}$$

and

$$\tilde{U} = \begin{bmatrix} U & 0 \\ (1 - β)U(I - βA)^{-1} & -(1 - β) \end{bmatrix}, \quad \tilde{y}_t = \begin{bmatrix} y_t \\ c_t \end{bmatrix}$$

Then we can express equation (11) as

$$x_{t+1} = \tilde{A} x_t + \tilde{C} w_{t+1}$$
$$\tilde{y}_t = \tilde{U} x_t$$

We can use the following formulas from linear state space models to compute population mean $μ_t = \mathbb{E} x_t$ and covariance $Σ_t := \mathbb{E}[(x_t - μ_t)(x_t - μ_t)']$

$$μ_{t+1} = \tilde{A} μ_t \quad \text{with} \quad μ_0 \text{ given}$$

$$Σ_{t+1} = \tilde{A} Σ_t \tilde{A}' + \tilde{C} \tilde{C}' \quad \text{with} \quad Σ_0 \text{ given}$$

We can then compute the mean and covariance of $\tilde{y}_t$ from

$$μ_{y,t} = \tilde{U} μ_t$$
$$Σ_{y,t} = \tilde{U} Σ_t \tilde{U}'$$
A Simple Example with iid Income

To gain some preliminary intuition on the implications of (11), let’s look at a highly stylized example where income is just iid.

(Later examples will investigate more realistic income streams)

In particular, let \( \{w_t\}_{t=1}^{\infty} \) be iid and scalar standard normal, and let

\[
\begin{align*}
    z_t &= \begin{bmatrix} z_t^1 \\ 1 \end{bmatrix}, \quad A = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 1 & \mu \end{bmatrix}, \quad C = \begin{bmatrix} \sigma \\ 0 \end{bmatrix}
\end{align*}
\]

Finally, let \( b_0 = z_0^1 = 0 \).

Under these assumptions we have \( y_t = \mu + \sigma w_t \sim N(\mu, \sigma^2) \).

Further, if you work through the state space representation, you will see that

\[
\begin{align*}
    b_t &= -\sigma \sum_{j=1}^{t-1} w_j \\
    c_t &= \mu + (1 - \beta) \sigma \sum_{j=1}^{t} w_j
\end{align*}
\]

Thus income is iid and debt and consumption are both Gaussian random walks.

Defining assets as \(-b_t\), we see that assets are just the cumulative sum of unanticipated incomes prior to the present date.

The next figure shows a typical realization with \( r = 0.05 \), \( \mu = 1 \), and \( \sigma = 0.15 \)

In [3]: using Plots, Random
\[
\begin{align*}
gr(fmt=:png);
\end{align*}
\]

Random.seed!(42)

\[
\begin{align*}
    \text{const } r &= 0.05 \\
    \text{const } \beta &= 1.0 / (1.0 + r) \\
    \text{const } T &= 60 \\
    \text{const } \sigma &= 0.15 \\
    \text{const } \mu &= 1.0
\end{align*}
\]

function time_path2()
    w = randn(T+1)
    w[1] = 0.0
    b = zeros(T+1)
    for t=2:T+1
        b[t] = sum(w[1:t])
    end
    \text{b .*= -}\sigma
    c = \mu .+ (1.0 .- \beta) .* (\sigma .* w .- b)
    return w, b, c
end

w, b, c = time_path2()

p = plot(0:T, \mu .+ \sigma .* w, color = :green, label = "non-financial income")
plot!(c, color = :black, label = "consumption")
Observe that consumption is considerably smoother than income.

The figure below shows the consumption paths of 250 consumers with independent income streams

```python
In [4]: time_paths = []
n = 250

for i in 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))
```

Out[4]:

![Graph showing consumption paths](image)
35.4 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.

35.4.1 Hall’s Representation

Hall [36] suggested an insightful way to summarize the implications of LQ permanent income theory.

First, to represent the solution for $b_t$, shift (9) forward one period and eliminate $b_{t+1}$ by using (2) 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

$$c_{t+1} - c_t = (1 - \beta) \sum_{j=0}^{\infty} \beta^j \{\mathbb{E}_{t+1}[y_{t+j+1}] - \mathbb{E}_t[y_{t+j+1}]\}$$

(16)

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 (16) and (8), which we repeat:
Equation (17) asserts that the consumer’s debt due at \( t \) equals the expected present value of its endowment minus the expected present value of its consumption stream.

A high debt thus indicates a large expected present value of surpluses \( y_t - c_t \).

Recalling again our discussion on forecasting geometric sums, we have

\[
\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j} = U(I - \beta A)^{-1} z_t
\]

\[
\mathbb{E}_{t+1} \sum_{j=0}^{\infty} \beta^j y_{t+j+1} = U(I - \beta A)^{-1} z_{t+1}
\]

\[
\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j+1} = U(I - \beta A)^{-1} A z_t
\]

Using these formulas together with (3) and substituting into (16) and (17) gives the following representation for the consumer’s optimum decision rule:

\[
\begin{align*}
    c_{t+1} &= c_t + (1 - \beta) U(I - \beta A)^{-1} C w_{t+1} \\
    b_t &= U(I - \beta A)^{-1} z_t - \frac{1}{1 - \beta} c_t \\
    y_t &= U z_t \\
    z_{t+1} &= A z_t + C w_{t+1}
\end{align*}
\]  

Representation (18) 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 \((1 - \beta) U(I - \beta A)^{-1} C w_{t+1}\).
- This is a more explicit representation of the martingale result in (6).

### 35.4.2 Cointegration

Representation (18) reveals that the joint process \( \{c_t, b_t\} \) possesses the property that Engle and Granger [27] 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 Section ??.

Despite this, both \( c_t \) and \( b_t \) will be non-stationary because they have unit roots (see (11) 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 (18) we have
\[(1 - \beta)b_t + c_t = (1 - \beta)U(I - \beta A)^{-1}z_t \tag{19}\]

Hence the linear combination \((1 - \beta)b_t + c_t\) is asymptotically stationary.

Accordingly, Granger and Engle would call \([(1 - \beta) 1]\) a **cointegrating vector** for the state.

When applied to the nonstationary vector process \([b_t \ c_t]'\), it yields a process that is asymptotically stationary.

Equation (19) can be rearranged to take the form

\[(1 - \beta)b_t + c_t = (1 - \beta)\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j}. \tag{20}\]

Equation (20) asserts that the **cointegrating residual** on the left side equals the conditional expectation of the geometric sum of future incomes on the right Section ??.

### 35.4.3 Cross-Sectional Implications

Consider again (18), this time in light of our discussion of distribution dynamics in the lecture on linear systems.

The dynamics of \(c_t\) are given by

\[c_{t+1} = c_t + (1 - \beta)U(I - \beta A)^{-1}Cw_{t+1} \tag{21}\]

or

\[c_t = c_0 + \sum_{j=1}^{t} \tilde{w}_j \quad \text{for} \quad \tilde{w}_{t+1} := (1 - \beta)U(I - \beta A)^{-1}Cw_{t+1}\]

The unit root affecting \(c_t\) causes the time \(t\) variance of \(c_t\) to grow linearly with \(t\).

In particular, since \(\{\tilde{w}_t\}\) is iid, we have

\[\text{Var}[c_t] = \text{Var}[c_0] + t \hat{\sigma}^2 \tag{22}\]

where

\[\hat{\sigma}^2 := (1 - \beta)^2U(I - \beta A)^{-1}CC'(I - \beta A')^{-1}U'\]

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 (22) 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., [21], [101]).
35.4.4 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 \( j \geq 1 \).

35.4.5 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) has the moving average representation

\[
y_{t+1} = d(L)w_{t+1}
\]

where

- \( d(L) = \sum_{j=0}^{\infty} d_j L^j \) for some sequence \( d_j \), where \( L \) is the lag operator Section ??
- at time \( t \), the consumer has an information set Section ?? \( w^t = [w_t, w_{t-1}, ...] \)

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

\[
\mathbb{E}_{t+1}[y_{t+j}] - \mathbb{E}_t[y_{t+j}] = d_{j-1} w_{t+1}
\]

Using (24) in (16) gives

\[
c_{t+1} - c_t = (1 − \beta) d(\beta) w_{t+1}
\]

The object \( d(\beta) \) is the present value of the moving average coefficients in the representation for the endowment process \( y_t \).

35.5 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 × 1 process distributed as \( N(0, I) \)
- \( z_{1t} \) is a permanent component of \( y_t \)
- \( z_{2t} \) is a purely transitory component of \( y_t \)
35.5. TWO CLASSIC EXAMPLES

35.5.1 Example 1

Assume as before that the consumer observes the state $z_t$ at time $t$.

In view of (18) we have

$$c_{t+1} - c_t = \sigma_1 w_{1t+1} + (1 - \beta)\sigma_2 w_{2t+1}$$

Formula (26) 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 (11) to this example shows that

$$b_{t+1} - b_t = -z_{2t} = -\sigma_2 w_{2t}$$

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

```plaintext
In [5]:
const r = 0.05
const β = 1.0 / (1.0 + r)
const T2 = 20  # Time horizon
const S = 5    # Impulse date
const σ1 = 0.15
const σ2 = 0.15

function time_path(permanent = false)
    w1 = zeros(T2 + 1)
    w2 = similar(w1)
    b = similar(w1)
    c = similar(w1)
    if permanent == false
        w2[S + 2] = 1.0
    else
        w1[S + 2] = 1.0
    end
    for t=2:T2
        b[t+1] = b[t] - σ2 * w2[t]
        c[t+1] = c[t] + σ1 * w1[t+1] + (1 - β) * σ2 * w2[t+1]
    end
    return b, c
end
```

$L = 0.175$
35.5.2 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 (18).

The discussion in sections 2.9.1 and 2.11.3 of [68] shows that the pertinent state space representation for $y_t$ is

$$
\begin{bmatrix}
    y_{t+1} \\
    a_{t+1}
\end{bmatrix} =
\begin{bmatrix}
    1 & -(1-K) \\
    0 & 0
\end{bmatrix}
\begin{bmatrix}
    y_t \\
    a_t
\end{bmatrix} +
\begin{bmatrix}
    1 \\
    1
\end{bmatrix}a_{t+1}
$$

$$
y_t = \begin{bmatrix}
    1 & 0
\end{bmatrix}
\begin{bmatrix}
    y_t \\
    a_t
\end{bmatrix}
$$

where

- $K :=$ the stationary Kalman gain
35.6. FURTHER READING

• \( a_t := y_t - E[y_t \mid y_{t-1}, \ldots, y_0] \)

In the same discussion in [68] 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 (18) implies

\[
c_{t+1} - c_t = [1 - \beta(1 - K)]a_{t+1}
\]

where the endowment process can now be represented in terms of the univariate innovation to \( y_t \) as

\[
y_{t+1} - y_t = a_{t+1} - (1 - K)a_t
\]

Equation (29) indicates that the consumer regards

• fraction \( K \) of an innovation \( a_{t+1} \) to \( y_{t+1} \) as permanent
• fraction \( 1 - K \) as purely transitory

The consumer permanently increases his consumption by the full amount of his estimate of the permanent part of \( a_{t+1} \), but by only \((1 - \beta)\) times his estimate of the purely transitory part of \( a_{t+1} \).

Therefore, in total he permanently increments his consumption by a fraction \( K + (1 - \beta)(1 - K) = 1 - \beta(1 - K) \) of \( a_{t+1} \).

He saves the remaining fraction \( \beta(1 - K) \).

According to equation (29), the first difference of income is a first-order moving average.

Equation (28) asserts that the first difference of consumption is iid.

Application of formula to this example shows that

\[
b_{t+1} - b_t = (K - 1)a_t
\]

This indicates how the fraction \( K \) of the innovation to \( y_t \) that is regarded as permanent influences the fraction of the innovation that is saved.

35.6 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’s 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., [37], [84], [20], [15].
35.7 Appendix: the Euler Equation

Where does the first order condition (5) come from?

Here we’ll give a proof for the two period case, which is representative of the general argument.

The finite horizon equivalent of the no-Ponzi condition is that the agent cannot end her life in debt, so \( b_2 = 0 \).

From the budget constraint (2) 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 \( u(c_0) + \beta E_0[u(c_1)] \) gives

\[
    \max_{b_1} \left\{ u \left( \frac{b_1}{R} - b_0 + y_0 \right) + \beta E_0[u(y_1 - b_1)] \right\}
\]

You will be able to verify that the first order condition is

\[
    u'(c_0) = \beta R E_0[u'(c_1)]
\]

Using \( \beta R = 1 \) gives (5) in the two period case.

The proof for the general case is similar.

Footnotes

[1] A linear marginal utility is essential for deriving (6) from (5). Suppose instead that we had imposed the following more standard assumptions on the utility function: \( u'(c) > 0, u''(c) < 0, u'''(c) > 0 \) and required that \( c \geq 0 \). The Euler equation remains (5). But the fact that \( u''' < 0 \) implies via Jensen’s inequality that \( E_t[u'(c_{t+1})] > u'(E_t[c_{t+1}]) \). This inequality together with (5) implies that \( E_t[c_{t+1}] > c_t \) (consumption is said to be a ‘submartingale’), so that consumption stochastically diverges to \( +\infty \). The consumer’s savings also diverge to \( +\infty \).

[2] An optimal decision rule is a map from current state into current actions—in this case, consumption.

[3] Representation (3) implies that \( d(L) = U(I - AL)^{-1}C \).

[4] This would be the case if, for example, the spectral radius of \( A \) is strictly less than one.

[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.

[6] See [58], [65], [66] for interesting applications of related ideas.
Chapter 36

Optimal Savings II: LQ Techniques

36.1 Contents

- Overview 36.2
- Introduction 36.3
- The LQ Approach 36.4
- Implementation 36.5
- Two Example Economies 36.6

Co-authored with Chase Coleman.

36.2 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 [36] 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 [8] to interpret intertemporal comovements of a government’s tax collections, its expenditures net of debt service, and its public debt.

This isomorphism means that in analyzing the LQ permanent income model, we are in effect also analyzing the Barro tax smoothing model.

It is just a matter of appropriately relabeling the variables in Hall’s model.

In this lecture, we’ll

- show how the solution to the LQ permanent income model can be obtained using LQ control methods
• represent the model as a linear state space system as in this lecture
• apply QuantEcon’s LSS type to characterize statistical features of the consumer’s optimal consumption and borrowing plans

We’ll then use these characterizations to construct a simple model of cross-section wealth and consumption dynamics in the spirit of Truman Bewley [13].

(Later we’ll study other Bewley models—see this lecture)

The model will prove useful for illustrating concepts such as
• stationarity
• ergodicity
• ensemble moments and cross section observations

### 36.2.1 Setup

**In [1]:** using InstantiateFromURL
# optionally add arguments to force installation: instantiate = true, precompile = true
github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

**In [2]:** using LinearAlgebra, Statistics

### 36.3 Introduction

Let’s recall the basic features of the model discussed in permanent income model. Consumer preferences are ordered by

$$E_0 \sum_{t=0}^{\infty} \beta^t u(c_t)$$

(1)

where \(u(c) = -(c - \gamma)^2\).

The consumer maximizes (1) by choosing a consumption, borrowing plan \(\{c_t, b_{t+1}\}_{t=0}^{\infty}\) subject to the sequence of budget constraints

$$c_t + b_t = \frac{1}{1+r} b_{t+1} + y_t, \quad t \geq 0$$

(2)

and the no-Ponzi condition

$$E_0 \sum_{t=0}^{\infty} \beta^t b_t^2 < \infty$$

(3)

The interpretation of all variables and parameters are the same as in the previous lecture.

We continue to assume that \((1+r)\beta = 1\).

The dynamics of \(\{y_t\}\) again follow the linear state space model

\[
\begin{align*}
z_{t+1} &= A z_t + C w_{t+1} \\
y_t &= U z_t
\end{align*}
\]
The restrictions on the shock process and parameters are the same as in our previous lecture.

### 36.3.1 Digression on a useful isomorphism

The LQ permanent income model of consumption is mathematically isomorphic with a version of Barro’s [8] 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 Barro’s model to consumption $c_t$ in the LQ permanent income model
- $G_t$, exogenous government expenditures in Barro’s model to nonfinancial income $y_t$ in the permanent income model
- $B_t$, government risk-free one-period assets falling due in Barro’s 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 Barro’s model to the gross rate of return $1 + r$ on financial assets in the permanent income model of consumption

then the two models are mathematically equivalent.

All characterizations of a $\{c_t, y_t, b_t\}$ in the LQ permanent income model automatically apply to a $\{T_t, G_t, B_t\}$ process in the Barro model of tax smoothing.

See consumption and tax smoothing models for further exploitation of an isomorphism between consumption and tax smoothing models.

### 36.3.2 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 (4), as discussed in our lecture on linear models.

To do so we take

$$z_t = \begin{bmatrix} 1 \\ y_t \\ y_{t-1} \end{bmatrix}, \quad A = \begin{bmatrix} 1 & 0 & 0 \\ \alpha & \rho_1 & \rho_2 \\ 0 & 1 & 0 \end{bmatrix}, \quad C = \begin{bmatrix} 0 \\ \sigma \end{bmatrix}, \quad \text{and} \quad U = [0 \ 1 \ 0]$$
36.4 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 consumer’s expectations about future nonfinancial income.

On the other hand, formulating the model in terms of an LQ dynamic programming problem reminds us that

- finding the state (of a dynamic programming problem) is an art, and
- iterations on a Bellman equation implicitly jointly solve both a forecasting problem and a control problem

36.4.1 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

\[
x_{t+1} = \tilde{A} x_t + \tilde{B} u_t + \tilde{C} w_{t+1}, \quad t \geq 0,
\]

(5)

where \( w_{t+1} \) is iid with mean vector zero and \( \mathbb{E} w_t w'_t = I \).

The tildes in \( \tilde{A}, \tilde{B}, \tilde{C} \) are to avoid clashing with notation in (4).

The value function for this problem is \( v(x) = -x'Px - d \), where

- \( P \) is the unique positive semidefinite solution of the corresponding matrix Riccati equation.
- The scalar \( d \) is given by \( d = \beta(1 - \beta)^{-1} \text{trace}(P \tilde{C} \tilde{C}') \).

The optimal policy is \( u_t = -Fx_t \), where \( F := \beta(Q + \beta \tilde{B}' \tilde{P} \tilde{B})^{-1} \tilde{B}' \tilde{P} \tilde{A} \).

Under an optimal decision rule \( F \), the state vector \( x_t \) evolves according to \( x_{t+1} = (\tilde{A} - \tilde{B} F)x_t + \tilde{C} w_{t+1} \).

36.4.2 Mapping into the LQ framework

To map into the LQ framework, we’ll use
\[ x_t := \begin{bmatrix} z_t \\ b_t \end{bmatrix} = \begin{bmatrix} 1 \\ y_t \\ y_{t-1} \\ b_t \end{bmatrix} \]

as the state vector and \( u_t := c_t - \gamma \) as the control.

With this notation and \( U_\gamma := \begin{bmatrix} \gamma & 0 & 0 \end{bmatrix} \), we can write the state dynamics as in (5) when

\[
\bar{A} := \begin{bmatrix} A & 0 \\ (1 + r)(U_\gamma - U) & 1 + r \end{bmatrix} \quad \bar{B} := \begin{bmatrix} 0 \\ 1 + r \end{bmatrix} \quad \text{and} \quad \bar{C} := \begin{bmatrix} C \\ 0 \end{bmatrix} \quad w_{t+1}
\]

Please confirm for yourself that, with these definitions, the LQ dynamics (5) match the dynamics of \( z_t \) and \( b_t \) described above.

To map utility into the quadratic form \( x'_t R x_t + u'_t Qu_t \) we can set

- \( Q := 1 \) (remember that we are minimizing) and
- \( R := \) a 4 \times 4 matrix of zeros

However, there is one problem remaining.

We have no direct way to capture the non-recursive restriction (3) on the debt sequence \( \{b_t\} \) from within the LQ framework.

To try to enforce it, we’re going to use a trick: put a small penalty on \( b_t^2 \) in the criterion function.

In the present setting, this means adding a small entry \( \epsilon > 0 \) in the (4, 4) position of \( R \).

That will induce a (hopefully) small approximation error in the decision rule.

We’ll check whether it really is small numerically soon.

### 36.5 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.

#### 36.5.1 The exogenous noinfinancial income process

First we create the objects for the optimal linear regulator

```julia
In [3]: using QuantEcon, LinearAlgebra
    using Plots
    gr(fmt=:png);
    # Set parameters
\( \alpha, \beta, \rho_1, \rho_2, \sigma = 10.0, 0.95, 0.9, 0.0, 1.0 \)

\( R = 1 / \beta \)
\( A = \begin{bmatrix} 1.0 & 0.0 & 0.0 \\ \alpha \rho_1 & \rho_2 \\ 0.0 & 1.0 & 0.0 \end{bmatrix} \)
\( C = \begin{bmatrix} 0.0 ; \sigma ; 0.0 \end{bmatrix}' \)
\( G = \begin{bmatrix} 0.0 & 1.0 & 0.0 \end{bmatrix} \)

\# Form LinearStateSpace system and pull off steady state moments
\( \mu_z_0 = \begin{bmatrix} 1.0, 0.0, 0.0 \end{bmatrix} \)
\( \Sigma_z_0 = \text{zeros}(3, 3) \)
\( Lz = \text{LSS}(A, C, G, \mu_0=\mu_z_0, \Sigma_0=\Sigma_z_0) \)
\( \mu_z, \mu_y, \Sigma_z, \Sigma_y = \text{stationary_distributions}(Lz) \)

\# 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.
\( mxbewley = mxo \)
\( sxbewley = sxo \)

\textbf{Out[3]:} \( 4 \times 4 \) Array{Float64,2}:

\[
\begin{bmatrix}
0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 4.73684 & 0.0 & 0.0 \\
0.0 & 4.73684 & 5.26316 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 \\
\end{bmatrix}
\]

The next step is to create the matrices for the LQ system

\textbf{In [4]:} \( A12 = \text{zeros}(3, 1) \)
\( ALQ_1 = \text{hcat}(A, A12) \)
\( ALQ_r = [0 \ -R \ 0 \ R] \)
\( ALQ = \text{vcat}(ALQ_1, ALQ_r) \)
\( RLQ = [0.0 \ 0.0 \ 0.0 \ 0.0 \ 0.0 \\
0.0 \ 0.0 \ 0.0 \ 0.0 \ 0.0 \\
0.0 \ 0.0 \ 0.0 \ 1e-9] \)
\( QLQ = 1.0 \)
\( BLQ = [0.0; 0.0; 0.0; R] \)
\( CLQ = [0.0; \sigma; 0.0; 0.0] \)
\( \beta_LQ = \beta \)

\textbf{Out[4]:} \( 0.95 \)

Let’s print these out and have a look at them.
36.5. **IMPLEMENTATION**

In [5]: println("A = \$ALQ")
println("B = \$BLQ")
println("R = \$RLQ")
println("Q = \$QLQ")

\[
A = \begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0; 10.0 & 0.9 & 0.0 & 0.0; 0.0 & 1.0 & 0.0 & 0.0; 0.0 & 0.0 & 0.0 & -1.0526315789473684 & 0.0 \\
1.0526315789473684 & 0.0 \end{bmatrix}
\]

\[
B = [0.0, 0.0, 0.0, 1.0526315789473684]
\]

\[
R = \begin{bmatrix} 0.0 & 0.0 & 0.0 & 0.0; 0.0 & 0.0 & 0.0 & 0.0; 0.0 & 0.0 & 0.0 & 0.0; 0.0 & 0.0 & 0.0 & 1.0e-9 \end{bmatrix}
\]

\[
Q = 1.0
\]

Now create the appropriate instance of an LQ model

In [6]: LQPI = QuantEcon.LQ(QLQ, RLQ, ALQ, BLQ, CLQ, bet=β_LQ);

We’ll save the implied optimal policy function soon and compare with what we get by employing an alternative solution method.

In [7]: P, F, d = stationary_values(LQPI)  # compute value function and decision rule
ABF = ALQ - BLQ * F  # form closed loop system

Out[7]: 4×4 Array{Float64,2}:
1.0  0.0  0.0  0.0
10.0 0.9  0.0  0.0
0.0  1.0  0.0  0.0
68.9655 -0.689655 0.0  1.0

### 36.5.2 Comparison with the difference equation approach

In our first lecture on the infinite horizon permanent income problem we used a different solution method.

The method was based around

- deducing the Euler equations that are the first-order conditions with respect to consumption and savings
- using the budget constraints and boundary condition to complete a system of expectational linear difference equations
- solving those equations to obtain the solution

Expressed in state space notation, the solution took the form

\[
\begin{align*}
z_{t+1} &= Az_t + Cw_{t+1} \\
b_{t+1} &= b_t + U[(I - \beta A)^{-1}(A - I)]z_t \\
y_t &= Uz_t \\
c_t &= (1 - \beta)[U(I - \beta A)^{-1}z_t - b_t]
\end{align*}
\]

Now we’ll apply the formulas in this system
\[ b_{pol} = G \ast (\text{inv}(I - \beta \ast A)) \ast (A - I) \]
\[ c_{pol} = (1 - \beta) \ast (G \ast \text{inv}(I - \beta \ast A)) \]

# 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
\[ \mu_0 = [1.0, 0.0, 0.0, 0.0] \]
\[ \Sigma_0 = \text{zeros}(4, 4) \]

A_LSS calculated as we have here should equal ABF calculated above using the LQ model

\[ \text{Out[8]: } 4\times4 \text{ Array}\{\text{Float64,2}\}: \]
\[
\begin{bmatrix}
0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 \\
\end{bmatrix}
\]

Now compare pertinent elements of \( c_{pol} \) and \( F \)

\[ \text{Out[9]: } 4\times4 \text{ Array}\{\text{Float64,2}\}: \]
\[
\begin{bmatrix}
0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 \\
-9.51248e-6 & 9.51248e-6 & 0.0 & -2.0e-8 \\
\end{bmatrix}
\]

We have verified that the two methods give the same solution.

Now let’s create instances of the LSS type and use it to do some interesting experiments. To do this, we’ll use the outcomes from our second method.
36.6 Two Example Economies

In the spirit of Bewley models [13], we’ll 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 \textit{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 \textit{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.

36.6.1 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.

\begin{verbatim}
In [11]: lss = LSS(A_LSS, C_LSS, G_LSS, mu_0=mu_0, Sigma_0=Sigma_0);
\end{verbatim}

36.6.2 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

\begin{verbatim}
In [12]: function income_consumption_debt_series(A, C, G, mu_0, Sigma_0, T = 150, npaths = 25)
    lss = LSS(A, C, G, mu_0=mu_0, Sigma_0=Sigma_0)
    # simulation/Moment Parameters
    moment_generator = moment_sequence(lss)
\end{verbatim}
# simulate various paths
bsim = zeros(npaths, T)
csim = zeros(npaths, T)
ysim = zeros(npaths, T)

for i in 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 = zeros(T)
cons_var = similar(cons_mean)
debt_mean = similar(cons_mean)
debt_var = similar(cons_mean)

for (idx, t) = enumerate(moment_generator)
    (μ_x, μ_y, Σ_x, Σ_y) = t
    cons_mean[idx], cons_var[idx] = μ_y[2], Σ_y[2, 2]
    debt_mean[idx], debt_var[idx] = μ_x[4], Σ_x[4, 4]

    if idx == T
        break
    end
end

return bsim, csim, ysim, cons_mean, cons_var, debt_mean, debt_var

function consumption_income_debt_figure(bsim, csim, ysim)
    # get T
    T = size(bsim, 2)

    # create first figure
    xvals = 1:T

    # plot consumption and income
    plt_1 = plot(csim[:, :], label="c", color=:blue, lw=2)
    plot!(plt_1, ysim[:, :], label="y", color=:green, lw=2)
    plot!(plt_1, csim', alpha=0.1, color=:blue, label="")
    plot!(plt_1, ysim', alpha=0.1, color=:green, label="")
    plot!(plt_1, title="Nonfinancial Income, Consumption, and Debt",
          xlabel="t", ylabel="y and c", legend=:bottomright)

    # plot debt
    plt_2 = plot(bsim[:, :], label="b", color=:red, lw=2)
    plot!(plt_2, bsim', alpha=0.1, color=:red, label="")
    plot!(plt_2, t', label="debt", legend=:bottomright)

    plot(plt_1, plt_2, layout=(2,1), size=(800,600))
end

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)
36.6. TWO EXAMPLE ECONOMIES

\[
c95 = 1.96 \times \sqrt{\text{cons}\_\text{var}}
\]
\[
c\_\text{perc}\_\text{95p}, c\_\text{perc}\_\text{95m} = \text{cons}\_\text{mean} + c95, \text{cons}\_\text{mean} - c95
\]
\[
c\_\text{perc}\_\text{90p}, c\_\text{perc}\_\text{90m} = \text{cons}\_\text{mean} + c90, \text{cons}\_\text{mean} - c90
\]

# create percentiles of cross-section distributions
\[
d\text{mean} = \text{mean}(\text{debt}\_\text{mean})
\]
\[
d90 = 1.65 \times \sqrt{\text{debt}\_\text{var}}
\]
\[
d95 = 1.96 \times \sqrt{\text{debt}\_\text{var}}
\]
\[
d\_\text{perc}\_\text{95p}, d\_\text{perc}\_\text{95m} = \text{debt}\_\text{mean} + d95, \text{debt}\_\text{mean} - d95
\]
\[
d\_\text{perc}\_\text{90p}, d\_\text{perc}\_\text{90m} = \text{debt}\_\text{mean} + d90, \text{debt}\_\text{mean} - d90
\]

xvals = 1:T

# first fanchart
plt_1 = plot(xvals, cons_mean, color=:black, lw=2, label="")
plot!(plt_1, xvals, Array(csim'), color=:black, alpha=0.25, label="")
plot!(xvals, fillrange=[c\_perc\_95m, c\_perc\_95p], alpha=0.25, color=:
  ↪ blue, label="")
plot!(xvals, fillrange=[c\_perc\_90m, c\_perc\_90p], alpha=0.25, color=:
  ↪ red, label="")
plot!(plt_1, title="Consumption/Debt over time",
  ylim=(cmean-15, cmean+15), ylabel="consumption")

# second fanchart
plt_2 = plot(xvals, debt_mean, color=:black, lw=2, label="")
plot!(plt_2, xvals, Array(bsim'), color=:black, alpha=0.25, label="")
plot!(xvals, fillrange=[d\_perc\_95m, d\_perc\_95p], alpha=0.25, color=:
  ↪ blue, label="")
plot!(xvals, fillrange=[d\_perc\_90m, d\_perc\_90p], alpha=0.25, color=:
  ↪ red, label="")
plot!(plt_2, ylabel="debt", xlabel="t")

plot(plt_1, plt_2, layout=(2,1), size=(800,600))

Out[12]: consumption_debt_fanchart (generic function with 1 method)

Now let's create figures with initial conditions of zero for \(y_0\) and \(b_0\)

In [13]: out = income_consumption_debt_series(A_LSS, C_LSS, G_LSS, \(\mu\_0, \Sigma\_0\))
bsim0, csim0, ysim0 = out[1:3]
cons\_mean0, cons\_var0, debt\_mean0, debt\_var0 = out[4:end]

consumption_income_debt_figure(bsim0, csim0, ysim0)

Out[13]:
In [14]: consumption_debt_fanchart(csim0, cons_mean0, cons_var0, bsim0, debt_mean0, debt_var0)

Out[14]:

Nonfinancial Income, Consumption, and Debt

Consumption/Debt over time
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.

\[ (1 - \beta) b_t + c_t = (1 - \beta) E_t \sum_{j=0}^{\infty} \beta^j y_{t+j} \]  

(6)

So at time 0 we have

\[ c_0 = (1 - \beta) E_0 \sum_{t=0}^{\infty} \beta^t y_t \]

This tells us that consumption starts at the income that would be paid by an annuity whose value equals the expected discounted value of nonfinancial income at time $t = 0$.

To support that level of consumption, the consumer borrows a lot early and consequently builds up substantial debt.

In fact, he or she incurs so much debt that eventually, in the stochastic steady state, he consumes less each period than his nonfinancial income.

He uses the gap between consumption and nonfinancial income mostly to service the interest payments due on his debt.

Thus, when we look at the panel of debt in the accompanying graph, we see that this is a group of 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 (6), 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 household’s present value of income and borrowing rise pretty much together.

This example illustrates the following point: the definition of cointegration implies that the cointegrating residual is asymptotically covariance stationary, not covariance stationary.

The cointegrating residual for the specification with zero income and zero debt initially has a notable transient component that dominates its behavior early in the sample.
By altering initial conditions, we shall remove this transient in our second example to be presented below.

In [15]:
   
   function cointegration_figure(bsim, csim)
       
       # create figure
       plot((1 - β) * bsim[1, :] + csim[1, :], color=:black, lw=2, label="")
       plot!(((1 - β) * bsim' + csim'), color=:black, alpha=.1, label="")
       plot!(title="Cointegration of Assets and Consumption", xlabel="t")
   
   end
   
   cointegration_figure(bsim0, csim0)

Out[15]:

36.6.3 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 man’s Bewley [13] model” in the following way

- as before, we start everyone at \(b_0 = 0\)
• But instead of starting everyone at \( y_{-1} = y_{-2} = 0 \), we draw \( \begin{bmatrix} y_{-1} \\ y_{-2} \end{bmatrix} \) from the invariant distribution of the \( \{y_t\} \) process.

This rigs a closed economy in which people are borrowing and lending with each other at a gross risk-free interest rate of \( R = \beta^{-1} \).

Across the group of people being analyzed, risk-free loans are in zero excess supply.

We have arranged primitives so that \( R = \beta^{-1} \) clears the market for risk-free loans at zero aggregate excess supply.

So the risk-free loans are being made from one person to another within our closed set of agent.

There is no need for foreigners to lend to our group.

Let’s have a look at the corresponding figures:

In [16]: 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)

Out[16]:

![Nonfinancial Income, Consumption, and Debt](image)

In [17]: consumption_debt_fanchart(csimb, cons_meanb, cons_varb, bsimb, debt_meanb, debt_varb)
The graphs confirm the following outcomes:

- As before, the consumption distribution spreads out over time.

But now there is some initial dispersion because there is *ex ante* heterogeneity in the initial draws of $\begin{bmatrix} y_{-1} \\ y_{-2} \end{bmatrix}$.

- As before, the cross-section distribution of debt spreads out over time.
- Unlike before, the average level of debt stays at zero, 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.

```python
In [18]: cointegration_figure(bsimb, csimb)
```

Out[18]:
Cointegration of Assets and Consumption
Chapter 37

Consumption and Tax Smoothing with Complete and Incomplete Markets

37.1 Contents

- Overview 37.2
- Background 37.3
- Model 1 (Complete Markets) 37.4
- Model 2 (One-Period Risk Free Debt Only) 37.5
- Example: Tax Smoothing with Complete Markets 37.6
- Linear State Space Version of Complete Markets Model 37.7

37.2 Overview

This lecture describes two types of consumption-smoothing and tax-smoothing models

- One is in the complete markets tradition of Lucas and Stokey [72].
- The other is in the incomplete markets tradition of Hall [36] and Barro [8].
- 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 [36] and Barro [8] 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 consumer’s intertemporal discount factor.

Barro [8] made an analogous assumption about the risk-free interest rate in a tax-smoothing model that we regard as isomorphic to Hall’s consumption-smoothing model.

We maintain Hall and Barro’s assumption about the interest rate when we describe an incomplete markets version of our model.

In addition, we extend their assumption about the interest rate to an appropriate counterpart
that we use in a “complete markets” model in the style of Lucas and Stokey [72].

While we are equally interested in consumption-smoothing and tax-smoothing models, for the most part we focus explicitly on consumption-smoothing versions of these models.

But for each version of the consumption-smoothing model there is a natural tax-smoothing counterpart obtained simply by

- relabeling consumption as tax collections and nonfinancial income as government expenditures
- relabeling the consumer’s debt as the government’s assets

For elaborations on this theme, please see Optimal Savings II: LQ Techniques and later parts of this lecture.

We’ll consider two closely related alternative assumptions about the consumer’s exogenous nonfinancial income process (or in the tax-smoothing interpretation, the government’s exogenous expenditure process):

- that it obeys a finite $N$ state Markov chain (setting $N = 2$ most of the time)
- that it is described by a linear state space model with a continuous state vector in $\mathbb{R}^n$ driven by a Gaussian vector iid shock process

We’ll spend most of this lecture studying the finite-state Markov specification, but will briefly treat the linear state space specification before concluding.

### 37.2.1 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 opt_tax_recur.

Linear-quadratic versions of the Lucas-Stokey tax-smoothing model are described in lqramsey.

The key differences between those lectures and this one are

- Here the decision maker takes all prices as exogenous, meaning that his decisions do not affect them.
- In lqramsey and opt_tax_recur, 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.

### 37.3 Background

Outcomes in consumption-smoothing (or tax-smoothing) models emerge from two sources:

- a decision maker – a consumer in the consumption-smoothing model or a government in the tax-smoothing model – who wants to maximize an intertemporal objective function that expresses its preference for paths of consumption (or tax collections) that are smooth in the sense of not varying across time and Markov states
- a set of trading opportunities that allow the optimizer to transform a possibly erratic nonfinancial income (or government expenditure) process into a smoother consumption (or tax collections) process by purchasing or selling financial securities
In the complete markets version of the model, each period the consumer can buy or sell one-period ahead state-contingent securities whose payoffs depend on next period’s realization of the Markov state.

In the two-state Markov chain case, there are two such securities each period.

In an $N$ state Markov state version of the model, $N$ 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 $\beta^{-1}$.

### 37.3.1 Finite State Markov Income Process

In each version of the consumption-smoothing model, nonfinancial income is governed by a two-state Markov chain (it’s easy to generalize this to an $N$ state Markov chain).

In particular, the state of the world is given by $s_t$ that follows a Markov chain with transition probability matrix

$$P_{ij} = P\{s_{t+1} = \bar{s}_j | s_t = \bar{s}_i\}$$

Nonfinancial income $\{y_t\}$ obeys

$$y_t = \begin{cases} \bar{y}_1 & \text{if } s_t = \bar{s}_1 \\ \bar{y}_2 & \text{if } s_t = \bar{s}_2 \end{cases}$$

A consumer wishes to maximize

$$\mathbb{E} \left[ \sum_{t=0}^{\infty} \beta^t u(c_t) \right] \quad \text{where} \quad u(c_t) = -(c_t - \gamma)^2 \quad \text{and} \quad 0 < \beta < 1$$

### Remark About Isomorphism

We can regard these as Barro [8] 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.

### 37.3.2 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
37.4 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 \( q(s_{t+1} \mid s_t) \) units of the time \( t \) consumption good.

At time \( t = 0 \), the consumer starts with an inherited level of debt due at time 0 of \( b_0 \) units of time 0 consumption goods.

The consumer’s budget constraint at \( t \geq 0 \) in Markov state \( s_t \) is

\[
c_t + b_t \leq y(s_t) + \sum_j q(\tilde{s}_j \mid s_t) b_{t+1}(\tilde{s}_j \mid s_t)
\]

where \( b_t \) is the consumer’s one-period debt that falls due at time \( t \) and \( b_{t+1}(\tilde{s}_j \mid s_t) \) are the consumer’s time \( t \) sales of the time \( t + 1 \) consumption good in Markov state \( \tilde{s}_j \), 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

\[
q(\tilde{s}_j \mid \tilde{s}_i) = \beta P_{ij}
\]

To understand this, observe that in state \( \tilde{s}_i \) it costs \( \sum_j q(\tilde{s}_j \mid \tilde{s}_i) \) to purchase one unit of consumption next period for sure, i.e., meaning no matter what state of the world occurs at \( t + 1 \).

Hence the implied price of a risk-free claim on one unit of consumption next period is

\[
\sum_j q(\tilde{s}_j \mid \tilde{s}_i) = \sum_j \beta P_{ij} = \beta
\]

This confirms that (2) 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 (2) on Arrow security prices,

\[
c_{t+1} = c_t
\]

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: We’ll make the plausible guess that

\[
b_{t+1}(\tilde{s}_j \mid s_t = \tilde{s}_i) = b(\tilde{s}_j), \quad i = 1, 2; \quad j = 1, 2
\]
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 \((4)\) about the consumer’s debt level choices.

For \( t \geq 1 \), these imply

\[
\begin{align*}
\bar{c} + b(\bar{s}_1) &= y(\bar{s}_1) + q(\bar{s}_1 | \bar{s}_1)b(\bar{s}_1) + q(\bar{s}_2 | \bar{s}_1)b(\bar{s}_2) \\
\bar{c} + b(\bar{s}_2) &= y(\bar{s}_2) + q(\bar{s}_1 | \bar{s}_2)b(\bar{s}_1) + q(\bar{s}_2 | \bar{s}_2)b(\bar{s}_2),
\end{align*}
\]

or

\[
\begin{bmatrix}
  b(\bar{s}_1) \\
  b(\bar{s}_2)
\end{bmatrix} + \begin{bmatrix}
  \bar{c} \\
  \bar{c}
\end{bmatrix} = \begin{bmatrix}
  y(\bar{s}_1) \\
  y(\bar{s}_2)
\end{bmatrix} + \beta \begin{bmatrix}
  P_{11} & P_{12} \\
  P_{21} & P_{22}
\end{bmatrix} \begin{bmatrix}
  b(\bar{s}_1) \\
  b(\bar{s}_2)
\end{bmatrix}
\]

These are 2 equations in the 3 unknowns \( \bar{c}, b(\bar{s}_1), b(\bar{s}_2) \).

To get a third equation, we assume that at time \( t = 0 \), \( b_0 \) is the debt due; and we assume that at time \( t = 0 \), the Markov state is \( \bar{s}_1 \).

Then the budget constraint at time \( t = 0 \) is

\[
\bar{c} + b_0 = y(\bar{s}_1) + q(\bar{s}_1 | \bar{s}_1)b(\bar{s}_1) + q(\bar{s}_2 | \bar{s}_1)b(\bar{s}_2)
\]

(6)

If we substitute (6) into the first equation of (5) and rearrange, we discover that

\[
b(\bar{s}_1) = b_0
\]

(7)

We can then use the second equation of (5) to deduce the restriction

\[
y(\bar{s}_1) - y(\bar{s}_2) + [q(\bar{s}_1 | \bar{s}_1) - q(\bar{s}_1 | \bar{s}_2) - 1]b_0 + [q(\bar{s}_2 | \bar{s}_1) + 1 - q(\bar{s}_2 | \bar{s}_2)]b(\bar{s}_2) = 0,
\]

(8)

an equation in the unknown \( b(\bar{s}_2) \).

Knowing \( b(\bar{s}_1) \) and \( b(\bar{s}_2) \), we can solve equation (6) for the constant level of consumption \( \bar{c} \).

### 37.4.1 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 \((4)\).
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.

### 37.4.2 Code

Here’s some code that, among other things, contains a function called consumption_complete().

This function computes $b(\tilde{s}_1), b(\tilde{s}_2), \tilde{c}$ as outcomes given a set of parameters, under the assumption of complete markets.

#### 37.4.3 Setup

```
In [1]: using InstantiateFromURL
     # optionally add arguments to force installation: instantiate = true,
     # precompile = true
     github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

In [2]: using LinearAlgebra, Statistics
     using Parameters, Plots, QuantEcon, Random
     gr(fmt = :png);

In [3]: ConsumptionProblem = @with_kw (β = 0.96,
     y = [2.0, 1.5],
     b0 = 3.0,
     P = [0.8 0.2
     0.4 0.6])

function consumption_complete(cp)
    @unpack β, P, y, b0 = cp  # Unpack
    y1, y2 = y                # extract income levels
    b1 = b0                  # b1 is known to be equal to b0
    Q = β * P                # assumed price system

    # Using equation (7) calculate b2
    b2 = (y2 - y1 - (Q[1, 1] - Q[2, 1] - 1) * b1) / (Q[1, 2] + 1 - Q[2, 2])

    # Using equation (5) calculate c
    c̃ = y1 - b0 + ([b1 b2] * Q[1, :])[1]

    return c̃, b1, b2
end

function consumption_incomplete(cp; N_simul = 150)
    @unpack β, P, y, b0 = cp  # unpack

    # for the simulation use the MarkovChain type
    mc = MarkovChain(P)

    # useful variables
```
37.5. MODEL 2 (ONE-PERIOD RISK FREE DEBT ONLY)

\( y = y \\
\mathbf{v} = \text{inv}(\mathbf{I} - \beta^* \mathbf{P}) \times y \)

# 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)
\( b_{\text{path}}[1] = b_0 \)

# optimal decisions from (12) and (13)
\( \mathbf{db} = ((1 - \beta)^* \mathbf{v} - y) / \beta \)

for \((i, s)\) in enumerate(s_path)
    \( c_{\text{path}}[i] = (1 - \beta)^*(\mathbf{v}[s, 1] - b_{\text{path}}[i]) \)
    \( b_{\text{path}}[i + 1] = b_{\text{path}}[i] + \mathbf{db}[s, 1] \)
end

return c_path, b_path[1:end - 1], y[s_path], s_path
end

Out[3]: consumption_incomplete (generic function with 1 method)

Let’s test by checking that \( c \) and \( b_2 \) satisfy the budget constraint

In [4]: cp = ConsumptionProblem()
    \( \mathbf{c}, \mathbf{b}_1, \mathbf{b}_2 = \text{consumption_complete}(cp) \)
    \( \text{debt_complete} = [\mathbf{b}_1, \mathbf{b}_2] \)
    \( \text{isapprox}((\mathbf{c} + \mathbf{b}_2 - \mathbf{c}.y[2] - \text{debt_complete}' \times (\mathbf{cp}.\beta \times \mathbf{cp}.P)[2, :])[1], 0) \)

Out[4]: true

Below, we’ll take the outcomes produced by this code – in particular the implied consumption and debt paths – and compare them with outcomes from an incomplete markets model in the spirit of Hall [36] and Barro [8] (and also, for those who love history, Gallatin (1807) [34]).

37.5 Model 2 (One-Period Risk Free Debt Only)

This is a version of the original models of Hall (1978) and Barro (1979) in which the decision maker’s ability to substitute intertemporally is constrained by his ability to buy or sell only one security, a risk-free one-period bond bearing a constant gross interest rate that equals \( \beta^{-1} \).

Given an initial debt \( b_0 \) at time 0, the consumer faces a sequence of budget constraints

\[ c_t + b_t = y_t + \beta b_{t+1}, \quad t \geq 0 \]

where \( \beta \) is the price at time \( t \) of a risk-free claim on one unit of time consumption at time \( t + 1 \).

First-order conditions for the consumer’s problem are
\[
\sum_j u'(c_{t+1,j})P_{ij} = u'(c_{t,i})
\]

For our assumed quadratic utility function this implies

\[
\sum_j c_{t+1,j}P_{ij} = c_{t,i},
\]

which is Hall’s (1978) conclusion that consumption follows a random walk.

As we saw in our first lecture on the permanent income model, this leads to

\[
b_t = \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j} - (1 - \beta)^{-1} c_t
\]

and

\[
c_t = (1 - \beta) \left[ \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j} - b_t \right].
\]

Equation (11) expresses \( c_t \) as a net interest rate factor \( 1 - \beta \) times the sum of the expected present value of nonfinancial income \( \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j} \) and financial wealth \(-b_t\).

Substituting (11) into the one-period budget constraint and rearranging leads to

\[
b_{t+1} - b_t = \beta^{-1} \left[ (1 - \beta) \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j} - y_t \right]
\]

Now let’s do a useful calculation that will yield a convenient expression for the key term \( \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j} \) in our finite Markov chain setting.

Define

\[
v_t := \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j}
\]

In our finite Markov chain setting, \( v_t = v(1) \) when \( s_t = \bar{s}_1 \) and \( v_t = v(2) \) when \( s_t = \bar{s}_2 \).

Therefore, we can write

\[
v(1) = y(1) + \beta P_{11} v(1) + \beta P_{12} v(2)
\]

\[
v(2) = y(2) + \beta P_{21} v(1) + \beta P_{22} v(2)
\]

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
In our finite Markov chain setting, from expression (11), consumption at date \( t \) when debt is \( b_t \) and the Markov state today is \( s_t = i \) is evidently

\[
c(b_t, i) = (1 - \beta) \left( [(I - \beta P)^{-1} \tilde{y}]_i - b_t \right)
\]

and the increment in debt is

\[
b_{t+1} - b_t = \beta^{-1} [(1 - \beta) v(i) - y(i)]
\]

37.5.1 Summary of Outcomes

In contrast to outcomes in the complete markets model, in the incomplete markets model

- consumption drifts over time as a random walk; the level of consumption at time \( t \) depends on the level of debt that the consumer brings into the period as well as the expected discounted present value of nonfinancial income at \( t \)
- the consumer's debt drifts upward over time in response to low realizations of nonfinancial income and drifts downward over time in response to high realizations of nonfinancial income
- the drift over time in the consumer’s debt and the dependence of current consumption on today’s debt level account for the drift over time in consumption

37.5.2 The Incomplete Markets Model

The code above also contains a function called consumption_incomplete() that uses (13) and (14) to

- simulate paths of \( y_t, c_t, b_{t+1} \)
- plot these against values of of \( \tilde{c}, b(s_1), b(s_2) \) found in a corresponding complete markets economy

Let’s try this, using the same parameters in both complete and incomplete markets economies
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 consumer’s debt oscillates between two values that are functions of the Markov state in the complete markets model, while the consumer’s debt drifts in a “unit root” fashion in the incomplete markets economy

**Using the Isomorphism**

We can simply relabel variables to acquire tax-smoothing interpretations of our two models.
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 [72], the government purchases insurance from the private sector.

- Purchasing insurance protects the government against the need to raise taxes too high or issue too much debt in the high government expenditure event.

We assume that government expenditures move between two values $G_1 < G_2$, where Markov state 1 means “peace” and Markov state 2 means “war”.

The government budget constraint in Markov state $i$ is

$$T_i + b_i = G_i + \sum_j Q_{ij}b_j$$
where

\[ Q_{ij} = \beta P_{ij} \]

is the price of one unit of output next period in state \( j \) when today’s Markov state is \( i \) and \( b_i \)
is the government’s level of assets in Markov state \( i \).

That is, \( b_i \) is the amount of the one-period loans owned by the government that fall due at
time \( t \).

As above, we’ll assume that the initial Markov state is state 1.

In addition, to simplify our example, we’ll set the government’s initial asset level to 0, so that
\( b_i = 0 \).

Here’s our code to compute a quantitative example with zero debt in peace time:

In [7]: # Parameters

\[
\beta = .96 \\
y = [1.0, 2.0] \\
b0 = 0.0 \\
\]

\[
P = [0.8 \ 0.2; \\
0.4 \ 0.6] \\
\]

\[
cp = ConsumptionProblem(\beta, y, b0, P) \\
Q = \beta * P \\
N_simul = 150 \\
\]

\[
\tilde{c}, b1, b2 = consumption_complete(cp) \\
debt_complete = [b1, b2] \\
\]

println("P = \$P") \\
println("Q = \$Q") \\
println("Govt expenditures in peace and war = \$y") \\
println("Constant tax collections = \$\tilde{c}") \\
println("Govt assets in two states = \$debt_complete") \\

msg = ""

Now let's check the government's budget constraint in peace and war.
Our assumptions imply that the government always purchases 0 units of the Arrow peace security.

println(msg) \\

\[
AS1 = Q[1, 2] * b2 \\
println("Spending on Arrow war security in peace = \$AS1") \\
AS2 = Q[2, 2] * b2 \\
println("Spending on Arrow war security in war = \$AS2") \\
\]

println("\n")

println("Government tax collections plus asset levels in peace and war") \\
TB1 = \tilde{c} + b1 \\
println("T+b in peace = \$TB1") \\
TB2 = \tilde{c} + b2 \\
println("T+b in war = \$TB2")
println("\n")
println("Total government spending in peace and war")
G1 = y[1] + AS1
G2 = y[2] + AS2
println("total govt spending in peace = $G1")
println("total govt spending in war = $G2")

println("\n")
println("Let's see ex post and ex ante returns on Arrow securities")

Π = 1 ./ Q  # reciprocal(Q)
exret = Π
println("Ex post returns to purchase of Arrow securities = $exret")
exant = Π .* P
println("Ex ante returns to purchase of Arrow securities = $exant")

P = [0.8 0.2; 0.4 0.6]
Q = [0.768 0.192; 0.384 0.576]
Govt expenditures in peace and war = [1.0, 2.0]
Constant tax collections = 1.3116883116883118
Govt assets in two states = [0.0, 1.6233766233766234]
Now let's check the government's budget constraint in peace and war.
Our assumptions imply that the government always purchases 0 units of the
Arrow peace security.

Spending on Arrow war security in peace = 0.3116883116883117
Spending on Arrow war security in war = 0.9350649350649349

Government tax collections plus asset levels in peace and war
T+b in peace = 1.3116883116883118
T+b in war = 2.9350649350649354

Total government spending in peace and war
total govt spending in peace = 1.3116883116883118
total govt spending in war = 2.935064935064935

Let's see ex post and ex ante returns on Arrow securities
Ex post returns to purchase of Arrow securities = [1.3020833333333333 5.208333333333333; 2.6041666666666665 1.7361111111111112]
Ex ante returns to purchase of Arrow securities = [1.0416666666666667 1.0416666666666667; 1.0416666666666667 1.0416666666666667]

37.6.1 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 con-
straints 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
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 $b_2 = -10$.

## 37.7 Linear State Space Version of Complete Markets Model

Now we’ll use a setting like that in first lecture on the permanent income model. In that model, there were

- incomplete markets: the consumer could trade only a single risk-free one-period bond bearing gross one-period risk-free interest rate equal to $\beta^{-1}$
- the consumer’s exogenous nonfinancial income was governed by a linear state space model driven by Gaussian shocks, the kind of model studied in an earlier lecture about linear state space models

We’ll write down a complete markets counterpart of that model.

So now we’ll suppose that nonfinancial income is governed by the state space system

\[
\begin{align*}
x_{t+1} &= A x_t + C w_{t+1} \\
y_t &= S y_t 
\end{align*}
\]

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

\[
p_{t+1}(x_{t+1} | x_t) = \beta \phi(x_{t+1} | A x_t, CC')
\]

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 (15), the value at $t$ of $b(x_{t+1})$ is

\[
\beta \int b(x_{t+1}) \phi(x_{t+1} | A x_t, CC') dx_{t+1} = \beta \mathbb{E}_t b_{t+1}
\]

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 consumer’s problem with complete markets and our assumption about Arrow securities prices is

\[ u'(c_{t+1}) = u'(c_t) \quad \text{for all } t \geq 0 \]

which again implies \( c_t = \bar{c} \) for some \( \bar{c} \).

So it follows that

\[ b_t = \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j (y_{t+j} - \bar{c}) \]

or

\[ b_t = S_y (I - \beta A)^{-1} x_t - \frac{1}{1 - \beta} \bar{c} \]

where the value of \( \bar{c} \) satisfies

\[ \bar{b}_0 = S_y (I - \beta A)^{-1} x_0 - \frac{1}{1 - \beta} \bar{c} \]

where \( \bar{b}_0 \) is an initial level of the consumer’s debt, specified as a parameter of the problem.

Thus, in the complete markets version of the consumption-smoothing model, \( c_t = \bar{c}, \forall t \geq 0 \) is determined by (17) and the consumer’s debt is a fixed function of the state \( x_t \) described by (16).

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.

In [8]: function complete_ss(\( \beta \), \( b_0 \), \( x_0 \), \( A \), \( C \), \( S_y \), \( T = 12 \))
# 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
# \( A_{\text{tilde}} = \text{vcat}(\text{hcat}(A, \text{zeros(size(A,1), 1)),
# \text{zeros(1, size(A,2) + 1)}) \)
# \( C_{\text{tilde}} = \text{vcat}(C, \text{zeros(1, 1)}) \)
# \( S_{y_{\text{tilde}}} = \text{hcat}(S_y, \text{zeros(1, 1)}) \)

\[
\text{lss} = \text{LSS}(A, C, S_y, \mu_0 = x0)
\]

# Add extra state to initial condition
# \( x0 = \text{hcat}(x0, 0) \)
# Compute the \( (I - \beta^* A)^{-1} \)
\[
\text{rm} = \text{inv}(I - \beta A)
\]

# Constant level of consumption
\[
c_{\text{bar}} = (1 - \beta) \times (S_y \times \text{rm} \times x0 .- b0)
\]

\[
c_{\text{hist}} = \text{ones}(T) \times c_{\text{bar}}[1]
\]

# Debt
\[
x_{\text{hist}}, y_{\text{hist}} = \text{simulate(lss, T)}
\]

\[
b_{\text{hist}} = (S_y \times \text{rm} \times x_{\text{hist}} .- c_{\text{bar}}[1] / (1.0 - \beta))
\]

\[
\text{return } c_{\text{hist}}, \text{vec}(b_{\text{hist}}), \text{vec}(y_{\text{hist}}), x_{\text{hist}}
\]

\[\text{end}\]

\[\text{N}_\text{simul} = 150\]

# Define parameters
\[\alpha, \rho_1, \rho_2 = 10.0, 0.9, 0.0\]
\[\sigma = 1.0\]

# \( N_\text{simul} = 1\)
# \( T = N_\text{simul}\)
\[A = [1.0 0.0 0.0; \alpha \rho_1 \rho_2; 0.0 1.0 0.0]\]
\[C = [0.0, \sigma, 0.0]\]
\[S_y = [1.0 1.0 0.0]\]
\[\beta, b0 = 0.95, -10.0\]
\[x0 = [1.0, \alpha / (1 - \rho_1), \alpha / (1 - \rho_1)]\]

# Do simulation for complete markets
\[\text{out} = \text{complete_ss}(\beta, b0, x0, A, C, S_y, 150)\]
\[c_{\text{hist}}_{\text{com}}, b_{\text{hist}}_{\text{com}}, y_{\text{hist}}_{\text{com}}, x_{\text{hist}}_{\text{com}} = \text{out}\]

# Consumption plots
\[\text{plt_cons} = \text{plot}(\text{title} = "Cons and income", \text{xlabel} = "Periods", \text{ylim} = [-5, 110])\]
\[\text{plot!(plt_cons, 1:N_simul, c_{\text{hist}}_{\text{com}}, \text{label} = "consumption", \text{lw} = 2)}\]
\[\text{plot!(plt_cons, 1:N_simul, y_{\text{hist}}_{\text{com}}, \text{label} = "income", \text{lw} = 2, \text{alpha} = 0.6, \text{linestyle} = :dash)}\]

# Debt plots
\[\text{plt_debt} = \text{plot}(\text{title} = "Debt and income", \text{xlabel} = "Periods")\]
\[\text{plot!(plt_debt, 1:N_simul, b_{\text{hist}}_{\text{com}}, \text{label} = "debt", \text{lw} = 2)}\]
\[\text{plot!(plt_debt, 1:N_simul, y_{\text{hist}}_{\text{com}}, \text{label} = "Income", \text{lw} = 2)}\]
37.7. LINEAR STATE SPACE VERSION OF COMPLETE MARKETS MODEL

\[ \text{lw} = 2, \text{alpha} = 0.6, \text{linestyle} = :\text{dash} \]
\[ \text{hline!}(\text{plt}\_\text{debt}, [0], \text{color} = :\text{black}, \text{linestyle} = :\text{dash}, \text{lw} = 2, \text{label} = "") \]
\[ \text{plot}(\text{plt}\_\text{cons}, \text{plt}\_\text{debt}, \text{layout} = (1,2), \text{size} = (800, 400)) \]
\[ \text{plot!}(\text{legend} = :\text{bottomleft}) \]

Out[8]:

### 37.7.1 Interpretation of Graph

In the above graph, please note that:

- nonfinancial income fluctuates in a stationary manner
- consumption is completely constant
- the consumer’s debt fluctuates in a stationary manner; in fact, in this case because nonfinancial income is a first-order autoregressive process, the consumer’s debt is an exact affine function (meaning linear plus a constant) of the consumer’s nonfinancial income

### 37.7.2 Incomplete Markets Version

The incomplete markets version of the model with nonfinancial income being governed by a linear state space system is described in the first lecture on the permanent income model and the followup lecture on the permanent income model.

In that version, consumption follows a random walk and the consumer’s debt follows a process with a unit root.

We leave it to the reader to apply the usual isomorphism to deduce the corresponding implications for a tax-smoothing model like Barro’s [8].

### 37.7.3 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.
Chapter 38

Optimal Savings III: Occasionally Binding Constraints

38.1 Contents

- Overview 38.2
- The Optimal Savings Problem 38.3
- Computation 38.4
- Exercises 38.5
- Solutions 38.6

38.2 Overview

Next we study an optimal savings problem for an infinitely lived consumer—the “common ancestor” described in [68], section 1.3.

This is an essential sub-problem for many representative macroeconomic models

- [1]
- [56]
- 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 [68]
- 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.
CHAPTER 38. OPTIMAL SAVINGS III: OCCASIONALLY BINDING CONSTRAINTS

38.2.1 References

Other useful references include [20], [22], [63], [87], [89] and [95].

38.3 The Optimal Savings Problem

Let’s write down the model and then discuss how to solve it.

38.3.1 Set Up

Consider a household that chooses a state-contingent consumption plan \( \{c_t\} \) to maximize

\[
\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t)
\]

subject to

\[
c_t + a_{t+1} \leq Ra_t + z_t, \quad c_t \geq 0, \quad a_t \geq -b \quad t = 0, 1, \ldots
\]  

(1)

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 (1), 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.

### 38.3.2 Value Function and Euler Equation

The value function \( V : S \to \mathbb{R} \) is defined by

\[
V(a, z) := \sup \{ \mathbb{E} \left[ \sum_{t=0}^{\infty} \beta^t u(c_t) \right] \}
\]  

(2)

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 (2).

To pin down such paths we can use a version of the Euler equation, which in the present setting is

\[
u'(c_t) \geq \beta R \mathbb{E}_t [u'(c_{t+1})]
\]

(3)

and

\[
u'(c_t) = \beta R \mathbb{E}_t [u'(c_{t+1})] \quad \text{whenever } c_t < Ra_t + z_t + b
\]

(4)

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) and (4) are equivalent to

\[
u'(c_t) = \max \{ \beta R \mathbb{E}_t [u'(c_{t+1})], u'(Ra_t + z_t + b) \}
\]

(5)

### 38.3.3 Optimality Results

Given our assumptions, it is known that

1. For each \((a, z) \in S\), a unique optimal consumption path from \((a, z)\) exists.
2. This path is the unique feasible path from \((a, z)\) satisfying the Euler equality (5) and the transversality condition.
$$\lim_{t \to \infty} \beta^t \mathbb{E}[u'(c_t)a_{t+1}] = 0.$$  

(6)

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 (5) and (6), and hence is the unique optimal path from $(a, z)$.

In summary, to solve the optimization problem, we need to compute $c^*$.

### 38.4 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.

#### 38.4.1 Time Iteration

We can rewrite (5) to make it a statement about functions rather than random variables. In particular, consider the functional equation

$$u' \circ c(a, z) = \max \left\{ \gamma \int u' \circ c \left\{ Ra + z - c(a, z), \hat{z} \right\} \Pi(z, d\hat{z}), \quad u'(Ra + z + b) \right\}$$

(7)

where $\gamma := \beta R$ and $u' \circ c(s) := u'(c(s))$.

Equation (7) 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

$$u'(t) = \max \left\{ \gamma \int u' \circ c \left\{ Ra + z - t, \hat{z} \right\} \Pi(z, d\hat{z}), \quad u'(Ra + z + b) \right\}$$

(8)

where

$$J(a, z) := \{ t \in \mathbb{R} : \min Z \leq t \leq Ra + z + b \}$$

(9)
We refer to \( K \) as Coleman’s policy function operator [17].

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 \( c \in \mathcal{C} \).

In particular, it can be shown that the path \( \{c_t\} \) generated from \( (a_0, z_0) \in S \) using policy function \( c^* \) is the unique optimal path from \( (a_0, z_0) \in S \).

**TL;DR** The unique optimal policy can be computed by picking any \( c \in \mathcal{C} \) and iterating with the operator \( K \) defined in (8).

### 38.4.2 Value Function Iteration

The Bellman operator for this problem is given by

\[
Tv(a, z) = \max_{0 \leq c \leq Ra + z + b} \left\{ u(c) + \beta \int v(Ra + z - c, 1)\Pi(z, d\hat{z}) \right\}
\]

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.

### 38.4.3 Implementation

Here’s the code for a named-tuple constructor called `ConsumerProblem` that stores primitives, as well as

- a \( T \) function, which implements the Bellman operator \( T \) specified above
- a \( K \) function, which implements the Coleman operator \( K \) specified above
- an `initialize`, which generates suitable initial conditions for iteration

### 38.4.4 Setup

In [1]:

```python
# optionally add arguments to force installation: instantiate = true,
# precompile = true
```
In [2]: using LinearAlgebra, Statistics
using BenchmarkTools, Optim, Parameters, Plots, QuantEcon, Random
using Optim: converged, maximum, maximizer, minimizer, iterations
gr(fmt = :png);

In [3]: # utility and marginal utility functions
u(x) = log(x)
du(x) = 1 / x

# model
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 = range(-b, grid_max, length = grid_size)
    return (r = r, R = R, β = β, b = b, Π = Π, z_vals = z_vals, asset_grid => asset_grid)
end

function T!(cp, V, out; ret_policy = false)
    # unpack input, set up arrays
    @unpack R, Π, β, b, asset_grid, z_vals = cp
    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 = maximize(obj, opt_lb, R .* a .+ z .+ b)
            converged(res) || error("Didn't converge")  # important to check
            if ret_policy
                out[i_a, i_z] = maximizer(res)
            else
                out[i_a, i_z] = maximum(res)
            end
        end
    end
end
T(cp, V; ret_policy = false) = 
T!(cp, V, similar(V); ret_policy = ret_policy)

get_greedy!(cp, V, out) = 
    update_bellman!(cp, V, out, ret_policy = true)

get_greedy(cp, V) = 
    update_bellman(cp, V, ret_policy = true)

function K!(cp, c, out)
    # simplify names, set up arrays
    @unpack R, Π, β, b, asset_grid, z_vals = cp
    z_idx = 1:length(z_vals)
    gam = R * β

    # policy function when the shock index is z_i
    cf = interp(asset_grid, c)

    # compute lower_bound for optimization
    opt_lb = 1e-8
    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] = minimizer(res)
        end
    end
    return out
end

K(cp, c) = K!(cp, c, similar(c))

function initialize(cp)
    # simplify names, set up arrays
    @unpack R, Π, β, b, asset_grid, z_vals = cp
    shape = length(asset_grid), length(z_vals)
    V, c = zeros(shape...), zeros(shape...)

    # populate V and c
    for (i_z, z) in enumerate(z_vals)
        for (i_a, a) in enumerate(asset_grid)
            c_max = R * a + z + b
            c[i_a, i_z] = c_max
            V[i_a, i_z] = u(c_max) / (1 - β)
        end
    end

    return V, c
Both $T$ and $K$ use linear interpolation along the asset grid to approximate the value and consumption functions.

The following exercises walk you through several applications where policy functions are computed.

In exercise 1 you will see that while VFI and TI produce similar results, the latter is much faster.

Intuition behind this fact was provided in a previous lecture on time iteration.

### 38.5 Exercises

#### 38.5.1 Exercise 1

The first exercise is to replicate the following figure, which compares TI and VFI as solution methods.

The figure shows consumption policies computed by iteration of $K$ and $T$ respectively.

- In the case of iteration with $T$, the final value function is used to compute the observed policy.

Consumption is shown as a function of assets with income $z$ held fixed at its smallest value.

The following details are needed to replicate the figure.
• The parameters are the default parameters in the definition of `consumerProblem`.
• The initial conditions are the default ones from `initialize(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
In [4]: cp = ConsumerProblem()
    v, c, = initialize(cp)
```

```Julia
Out[4]: ([[-17.328679513998615 0.0; -4.664387247760648 7.125637140580436; ... ; 69.82541010621486
      70.57937835479346; 70.32526591846735 71.06452735149534], [0.5 1.0; 0.8297959183673469
      1.329795918367347; ... ; 16.33020408163265 16.83020408163265; 16.66 17.16])
```

```Julia
In [5]: @btime T(cp, v);

530.441 μs (4676 allocations: 428.20 KiB)
```

```Julia
In [6]: @btime K(cp, c);

768.181 μs (7925 allocations: 742.17 KiB)
```

38.5.2 Exercise 2

Next let’s consider how the interest rate affects consumption.

Reproduce the following figure, which shows (approximately) optimal consumption policies for different interest rates
• Other than $r$, all parameters are at their default values
• $r$ steps through range(0, 0.04, length = 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.

### 38.5.3 Exercise 3

Now let’s consider the long run asset levels held by households.

We’ll take $r = 0.03$ and otherwise use default parameters.

The following figure is a 45 degree diagram showing the law of motion for assets when consumption is optimal

In [7]: # solve for optimal consumption

```python
m = ConsumerProblem(r = 0.03, grid_max = 4)
```

```python
v_init, c_init = initialize(m)
```

```python
c = compute_fixed_point(c -> K(m, c),
    c_init,
    max_iter = 150,
    verbose = false)
```

```python
a = m.asset_grid
```

```python
R, z_vals = m.R, m.z_vals
```

```python
# generate savings plot
```

```python
plot(a, R * a .+ z_vals[1] - c[:, 1], label = "Low income")
```

```python
plot!(xlabel = "Current assets", ylab = "Next period assets")
```

```python
plot!(a, R * a .+ z_vals[2] - c[:, 2], label = "High income")
```

```python
plot!(xlabel = "Current assets", ylab = "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 [54].
- Represents the long run dispersion of assets across households when households have idiosyncratic shocks.

Ergodicity is valid here, so stationary probabilities can be calculated by averaging over a single long time series.

- Hence to approximate the stationary distribution we can simulate a long time series for assets and histogram, as in the following figure
Your task is to replicate the figure

- Parameters are as discussed above
- The histogram in the figure used a single time series \( \{a_t\} \) of length 500,000
- Given the length of this time series, the initial condition \((a_0, z_0)\) will not matter
- You might find it helpful to use the `MarkovChain` type from `quantecon`

### 38.5.4 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: [68] 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.

### 38.6 Solutions

#### 38.6.1 Exercise 1

In [8]: 

```python
    cp = ConsumerProblem()
    N = 80

    V, c = initialize(cp)
    println("Starting value function iteration")
    for i in 1:N
        V = T(cp, V)
    end
c1 = T(cp, V, ret_policy=true)

    V2, c2 = initialize(cp)
    println("Starting policy function iteration")
    for i in 1:N
        c2 = K(cp, c2)
    end

    plot(cp.asset_grid, c1[:, 1], label = "value function iteration")
    plot!(cp.asset_grid, c2[:, 1], label = "policy function iteration")
```
38.6.2 Exercise 2

```
In [9]: r_vals = range(0, 0.04, length = 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 -> K(cp, x),
                                 c_init,
                                 max_iter = 150,
                                 verbose = false)
       traces = push!(traces, c[:, 1])
       legends = push!(legends, "r = $(round(r_val, digits = 3))")
   end

   plot(traces, label = reshape(legends, 1, length(legends)))
   plot!(xlabel = "asset level", ylabel = "Consumption (low income)")
   plot!(legend = :topleft)
```

Out[9]:
38.6.3 Exercise 3

In [10]: `function compute_asset_series(cp, T = 500_000; verbose = false)
   @unpack II, z_vals, R = cp  # simplify names
   z_idx = 1:length(z_vals)
   v_init, c_init = initialize(cp)
   c = compute_fixed_point(x -> K(cp, x), c_init,
                           max_iter = 150, verbose = false)

   cf = interp(cp.asset_grid, c)

   a = zeros(T + 1)
   z_seq = simulate(MarkovChain(II), T)
   for t in 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)
Random.seed!(42) # for reproducibility
a = compute_asset_series(cp)
histogram(a, nbins = 20, leg = false, normed = true, xlabel = "assets")

Out[10]:
38.6.4 Exercise 4

In [11]: M = 25
   ...: r_vals = range(0, 0.04, length = 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, 250_000))
   ...:         asset_mean[i] = the_mean
   ...:     xs = push!(xs, asset_mean)
   ...:     ys = 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]))
   ...: plot!(legend = :bottomright)

Finished iteration b = 1.0
Finished iteration b = 3.0

Out[11]:
Chapter 39

Robustness

39.1 Contents

- Overview 39.2
- The Model 39.3
- Constructing More Robust Policies 39.4
- Robustness as Outcome of a Two-Person Zero-Sum Game 39.5
- The Stochastic Case 39.6
- Implementation 39.7
- Application 39.8
- Appendix 39.9

39.2 Overview

This lecture modifies a Bellman equation to express a decision maker’s doubts about transition dynamics.

His specification doubts make the decision maker want a robust decision rule.

Robust means insensitive to misspecification of transition dynamics.

The decision maker has a single approximating model.

He calls it approximating to acknowledge that he doesn’t completely trust it.

He fears that outcomes will actually be determined by another model that he cannot describe explicitly.

All that he knows is that the actual data-generating model is in some (uncountable) set of models that surrounds his approximating model.

He quantifies the discrepancy between his approximating model and the genuine data-generating model by using a quantity called entropy.

(We’ll explain what entropy means below)

He wants a decision rule that will work well enough no matter which of those other models actually governs outcomes.

This is what it means for his decision rule to be “robust to misspecification of an approximating model”.

707
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.

### 39.2.1 Sets of Models Imply Sets Of Values

Our “robust” decision maker wants to know how well a given rule will work when he does not know a single transition law …

... he wants to know sets of values that will be attained by a given decision rule $F$ under a set of transition laws.

Ultimately, he wants to design a decision rule $F$ that shapes these sets of values in ways that he prefers.

With this in mind, consider the following graph, which relates to a particular decision problem to be explained below.
The figure shows a *value-entropy correspondence* for a particular decision rule $F$.

The shaded set is the graph of the correspondence, which maps entropy to a set of values associated with a set of models that surround the decision maker’s approximating model.

Here

- *Value* refers to a sum of discounted rewards obtained by applying the decision rule $F$ when the state starts at some fixed initial state $x_0$.
- *Entropy* is a nonnegative number that measures the size of a set of models surrounding the decision maker’s approximating model.
  - Entropy is zero when the set includes only the approximating model, indicating that the decision maker completely trusts the approximating model.
  - Entropy is bigger, and the set of surrounding models is bigger, the less the decision maker trusts the approximating model.

The shaded region indicates that for all models having entropy less than or equal to the number on the horizontal axis, the value obtained will be somewhere within the indicated set of values.

Now let’s compare sets of values associated with two different decision rules, $F_r$ and $F_b$.

In the next figure,

- The red set shows the value-entropy correspondence for decision rule $F_r$.
- The blue set shows the value-entropy correspondence for decision rule $F_b$.

The blue correspondence is skinnier than the red correspondence.

This conveys the sense in which the decision rule $F_b$ is *more robust* than the decision rule $F_r$.

- *more robust* means that the set of values is less sensitive to *increasing misspecification* as measured by entropy.
CHAPTER 39. 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 we’ll explain in detail how to construct these sets of values for a given $F$, but for now ....

Here is a hint about the secret weapons we’ll use to construct these sets

- We’ll use some min problems to construct the lower bounds.
- We’ll use some max problems to construct the upper bounds.

We will also describe how to choose $F$ to shape the sets of values.

This will involve crafting a skinnier set at the cost of a lower level (at least for low values of entropy).

39.2.2 Inspiring Video

If you want to understand more about why one serious quantitative researcher is interested in this approach, we recommend Lars Peter Hansen’s Nobel lecture.

39.2.3 Other References

Our discussion in this lecture is based on

- [45]
- [39]

39.2.4 Setup

In [1]: using InstantiateFromURL
       # optionally add arguments to force installation: instantiate = true, precompile = true
       github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

In [2]: using LinearAlgebra, Statistics

39.3 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
39.4. CONSTRUCTING MORE ROBUST POLICIES 711

\[
\sum_{t=0}^{\infty} \beta^t \{ x_t' R x_t + u_t' Q u_t \} \tag{1}
\]

subject to the linear law of motion

\[
x_{t+1} = A x_t + B u_t + C w_{t+1}, \quad t = 0, 1, 2, \ldots \tag{2}
\]

As before,

- \( x_t \) is \( n \times 1 \), \( A \) is \( n \times n \)
- \( u_t \) is \( k \times 1 \), \( B \) is \( n \times k \)
- \( w_t \) is \( j \times 1 \), \( C \) is \( n \times j \)
- \( R \) is \( n \times n \) and \( Q \) is \( k \times k \)

Here \( x_t \) is the state, \( u_t \) is the control, and \( w_t \) is a shock vector.

For now we take \( \{ w_t \} := \{ w_t \}_{t=1}^{\infty} \) to be deterministic — a single fixed sequence.

We also allow for model uncertainty on the part of the agent solving this optimization problem.

In particular, the agent takes \( w_t = 0 \) for all \( t \geq 0 \) as a benchmark model, but admits the possibility that this model might be wrong.

As a consequence, she also considers a set of alternative models expressed in terms of sequences \( \{ w_t \} \) that are “close” to the zero sequence.

She seeks a policy that will do well enough for a set of alternative models whose members are pinned down by sequences \( \{ w_t \} \).

Soon we’ll quantify the quality of a model specification in terms of the maximal size of the expression \( \sum_{t=0}^{\infty} \beta^{t+1} w_{t+1}' w_{t+1} \).

39.4 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

\[
J_{t-1}(x) = \min_u \{ x' R x + u' Q u + \beta J_t(A x + B u + C w_t) \}
\]

(Here \( J \) depends on \( t \) because the sequence \( \{ w_t \} \) is not recursive)

Our tool for studying robustness is to construct a rule that works well even if an adverse sequence \( \{ w_t \} \) occurs.

In our framework, “adverse” means “loss increasing”.

As we’ll see, this will eventually lead us to construct the Bellman equation.

\[
J(x) = \min_u \max_w \{ x' R x + u' Q u + \beta [J(A x + B u + C w) - \theta w' w] \}
\]

(3)

Notice that we’ve added the penalty term \(-\theta w' w\).

Since \( w' w = \| w \|^2 \), this term becomes influential when \( w \) moves away from the origin.
The penalty parameter \( \theta \) controls how much we penalize the maximizing agent for “harming” the minimizing agent.

By raising \( \theta \) more and more, we more and more limit the ability of maximizing agent to distort outcomes relative to the approximating model.

So bigger \( \theta \) is implicitly associated with smaller distortion sequences \( \{w_t\} \).

### 39.4.1 Analyzing the Bellman equation

So what does \( J \) in (3) look like?

As with the ordinary LQ control model, \( J \) takes the form \( J(x) = x'Px \) for some symmetric positive definite matrix \( P \).

One of our main tasks will be to analyze and compute the matrix \( P \).

Related tasks will be to study associated feedback rules for \( u_t \) and \( w_{t+1} \).

First, using matrix calculus, you will be able to verify that

\[
\max_w \{ (Ax + Bu + Cw)'P(Ax + Bu + Cw) - \theta w'w \} = (Ax + Bu)' \mathcal{D}(P)(Ax + Bu)
\]

where

\[
\mathcal{D}(P) := P + PC(\theta I - C'PC)^{-1}C'P
\]

and \( I \) is a \( j \times j \) identity matrix. Substituting this expression for the maximum into (3) yields

\[
x'Px = \min_u \{ x'Rx + u'Qu + \beta (Ax + Bu)' \mathcal{D}(P)(Ax + Bu) \}
\]

Using similar mathematics, the solution to this minimization problem is \( u = -Fx \) where

\[
F := (Q + \beta B' \mathcal{D}(P)B)^{-1}B' \mathcal{D}(P)A.
\]

Substituting this minimizer back into (6) and working through the algebra gives \( x'Px = x'\mathcal{B}(\mathcal{D}(P))x \) for all \( x \), or, equivalently,

\[
P = \mathcal{B}(\mathcal{D}(P))
\]

where \( \mathcal{D} \) is the operator defined in (5) and

\[
\mathcal{B}(P) := R - \beta^2 A'PB(Q + \beta B'PB)^{-1}B'PA + \beta A'PA
\]

The operator \( \mathcal{B} \) is the standard (i.e., non-robust) LQ Bellman operator, and \( P = \mathcal{B}(P) \) is the standard matrix Riccati equation coming from the Bellman equation — see this discussion.

Under some regularity conditions (see [39]), the operator \( \mathcal{B} \circ \mathcal{D} \) has a unique positive definite fixed point, which we denote below by \( \hat{P} \).

A robust policy, indexed by \( \theta \), is \( u = -\hat{F}x \) where

\[
\hat{F} := (Q + \beta B' \mathcal{D}(\hat{P})B)^{-1}B' \mathcal{D}(\hat{P})A
\]
39.5. **ROBUSTNESS AS OUTCOME OF A TWO-PERSON ZERO-SUM GAME**

We also define

\[
\hat{K} := (\theta I - C'\hat{P}C)^{-1}C'\hat{P}(A - B\hat{F})
\]  

(8)

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 (4) evaluated at the fixed rule \(u = -\hat{F}x\).

Note that \(\hat{P}, \hat{F}, \hat{K}\) are all determined by the primitives and \(\theta\).

Note also that if \(\theta\) is very large, then \(\mathcal{D}\) is approximately equal to the identity mapping.

Hence, when \(\theta\) is large, \(\hat{P}\) and \(\hat{F}\) are approximately equal to their standard LQ values.

Furthermore, when \(\theta\) is large, \(\hat{K}\) is approximately equal to zero.

Conversely, smaller \(\theta\) is associated with greater fear of model misspecification, and greater concern for robustness.

**39.5 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.

**39.5.1 Agent 2’s Problem**

Agent 2

1. knows a fixed policy \(F\) specifying the behavior of agent 1, in the sense that \(u_t = -Fx_t\) for all \(t\)

2. responds by choosing a shock sequence \(\{w_t\}\) from a set of paths sufficiently close to the benchmark sequence \(\{0, 0, 0, \ldots\}\)

A natural way to say “sufficiently close to the zero sequence” is to restrict the summed inner product \(\sum_{t=1}^{\infty} w'_tw_t\) to be small.

However, to obtain a time-invariant recursive formulation, it turns out to be convenient to restrict a discounted inner product

\[
\sum_{t=1}^{\infty} \beta^t w'_tw_t \leq \eta
\]  

(9)
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 (1), this value can be written as

$$J_F(x_0, w) := \sum_{t=0}^{\infty} \beta^t x'_t (R + F'QF)x_t$$

where

$$x_{t+1} = (A - BF)x_t + Cw_{t+1}$$

and the initial condition $x_0$ is as specified in the left side of (10).

Agent 2 chooses $w$ to maximize agent 1’s loss $J_F(x_0, w)$ subject to (9).

Using a Lagrangian formulation, we can express this problem as

$$\max_w \sum_{t=0}^{\infty} \beta^t \left\{ x'_t (R + F'QF)x_t - \beta \theta (w'_{t+1} w_{t+1} - \eta) \right\}$$

where \{x_t\} satisfied (11) and $\theta$ is a Lagrange multiplier on constraint (9).

For the moment, let’s take $\theta$ as fixed, allowing us to drop the constant $\beta \theta \eta$ term in the objective function, and hence write the problem as

$$\max_w \sum_{t=0}^{\infty} \beta^t \left\{ x'_t (R + F'QF)x_t - \beta \theta w'_{t+1} w_{t+1} \right\}$$

or, equivalently,

$$\min_w \sum_{t=0}^{\infty} \beta^t \left\{ -x'_t (R + F'QF)x_t + \beta \theta w'_{t+1} w_{t+1} \right\}$$

subject to (11).

What’s 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 2’s problem is to set $\theta$ to enforce the constraint (9), which can be done by choosing $\theta = \theta_\eta$ such that

$$\beta \sum_{t=0}^{\infty} \beta^t x'_t K(F, \theta_\eta)' K(F, \theta_\eta) x_t = \eta$$

Here $x_t$ is given by (11) — which in this case becomes $x_{t+1} = (A - BF + CK(F, \theta))x_t$. 

39.5. ROBUSTNESS AS OUTCOME OF A TWO-PERSON ZERO-SUM GAME

39.5.2 Using Agent 2’s Problem to Construct Bounds on the Value Sets

The Lower Bound

Define the minimized object on the right side of problem (12) as $R_\theta(x_0, F)$.

Because “minimizers minimize” we have

$$
R_\theta(x_0, F) \leq \sum_{t=0}^{\infty} \beta^t \{-x_0'([R + F'QF])x_t\} + \beta\theta \sum_{t=0}^{\infty} \beta^t w_{t+1} w_{t+1},
$$

where $x_{t+1} = (A - BF + CK(F, \theta))x_t$ and $x_0$ is a given initial condition.

This inequality in turn implies the inequality

$$
R_\theta(x_0, F) - \theta \text{ent} \leq \sum_{t=0}^{\infty} \beta^t \{-x_0'([R + F'QF])x_t\}
$$

(14)

where

$$
\text{ent} := \beta \sum_{t=0}^{\infty} \beta^t w_{t+1} w_{t+1}
$$

The left side of inequality (14) 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 (14) is attained when

$$
\text{ent} = \beta \sum_{t=0}^{\infty} \beta^t x_0'K(F, \theta)'K(F, \theta)x_t
$$

(15)

To construct the lower bound on the set of values associated with all perturbations $w$ satisfying the entropy constraint (9) at a given entropy level, we proceed as follows:

- For a given $\theta$, solve the minimization problem (12).
- Compute the minimizer $R_\theta(x_0, F)$ and the associated entropy using (15).
- 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 (12) with the maximization problem.

\[
V_\tilde{\theta}(x_0, F) = \max_w \sum_{t=0}^{\infty} \beta^t \left\{ -x'_t (R + F'QF)x_t - \beta \tilde{\theta} w'_{t+1} w_{t+1} \right\}
\]  

(16)

where now \(\tilde{\theta} > 0\) penalizes the choice of \(w\) with larger entropy.

(Notice that \(\tilde{\theta} = -\theta\) in problem (12))

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

\[
V_\tilde{\theta}(x_0, F) + \tilde{\theta} \text{ent} \geq \sum_{t=0}^{\infty} \beta^t \left\{ -x'_t (R + F'QF)x_t \right\}
\]  

(17)

where

\[
\text{ent} \equiv \beta \sum_{t=0}^{\infty} \beta^t w'_{t+1} w_{t+1}
\]

The left side of inequality (17) is a straight line with slope \(\tilde{\theta}\).

The upper bound on the left side of (17) is attained when

\[
\text{ent} = \beta \sum_{t=0}^{\infty} \beta^t x'_t K(F, \tilde{\theta})' K(F, \tilde{\theta}) x_t
\]  

(18)

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 (16).
- Compute the maximizer \(V_\tilde{\theta}(x_0, F)\) and the associated entropy using (18).
- Compute the upper bound on the value function \(V_\tilde{\theta}(x_0, F) + \tilde{\theta} \text{ent}\) and plot it against ent.
- Repeat the preceding three steps for a range of values of \(\tilde{\theta}\) to trace out the upper bound.

Reshaping the set of values

Now in the interest of reshaping these sets of values by choosing \(F\), we turn to agent 1’s problem.
39.5. ROBUSTNESS AS OUTCOME OF A TWO-PERSON ZERO-SUM GAME

39.5.3 Agent 1’s Problem

Now we turn to agent 1, who solves

$$\min_{\{u_t\}} \sum_{t=0}^{\infty} \beta^t \{x_t'Rx_t + u_t'Qu_t - \beta \theta w_{t+1}'w_{t+1}\}$$

where $\{w_{t+1}\}$ satisfies $w_{t+1} = Kx_t$.

In other words, agent 1 minimizes

$$\sum_{t=0}^{\infty} \beta^t \{x_t'(R - \beta \theta K'K)x_t + u_t'Qu_t\}$$

subject to

$$x_{t+1} = (A + CK)x_t + Bu_t$$

Once again, the expression for the optimal policy can be found here — we denote it by $\tilde{F}$.

39.5.4 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 (7) 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 (8)
2. $\Phi(\tilde{K}) = \tilde{F}$

A sketch of the proof is given in the appendix.
39.6 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) with the stochastic analogue

\[
J(x) = \min_u \max_{\psi \in \mathcal{P}} \left\{ x'R x + u'Q u + \beta \left( \int J(A x + B u + C w) \psi(dw) - \theta D_{KL}(\psi, \phi) \right) \right\}
\]

Here \( \mathcal{P} \) represents the set of all densities on \( \mathbb{R}^n \) and \( \phi \) is the benchmark distribution \( N(0, I) \).

The distribution \( \phi \) is chosen as the least desirable conditional distribution in terms of next period outcomes, while taking into account the penalty term \( \theta D_{KL}(\psi, \phi) \).

This penalty term plays a role analogous to the one played by the deterministic penalty \( \theta w'w \) in (3), since it discourages large deviations from the benchmark.

39.6.1 Solving the Model

The maximization problem in (22) 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 \( J(x) = x'P x + d \) for some positive definite matrix \( P \) and constant real number \( d \).

Moreover, it turns out that if \( (I - \theta^{-1}C'PC)^{-1} \) is nonsingular, then

\[
\max_{\psi \in \mathcal{P}} \left\{ \int (A x + B u + C w)' P(A x + B u + C w) \psi(dw) - \theta D_{KL}(\psi, \phi) \right\} = (A x + B u)' \mathcal{D}(P)(A x + B u) + \kappa(\theta, P)
\]
where

\[ \kappa(\theta, P) := \theta \ln[\det(I - \theta^{-1}C'PC)^{-1}] \]

and the maximizer is the Gaussian distribution

\[ \psi = N((\theta I - C'PC)^{-1}C'P(Ax + Bu), (I - \theta^{-1}C'PC)^{-1}) \] (24)

Substituting the expression for the maximum into Bellman equation (22) and using \( J(x) = x'Px + d \) gives

\[ x'Px + d = \min_u \{ x'Rx + u'Qu + \beta (Ax + Bu)'D(P)(Ax + Bu) + \beta [d + \kappa(\theta, P)] \} \] (25)

Since constant terms do not affect minimizers, the solution is the same as (6), 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

\[ \hat{d} := \frac{\beta}{1 - \beta}\kappa(\theta, P) \] (26)

The robust policy in this stochastic case is the minimizer in (25), which is once again \( u = -\hat{F}x \) for \( \hat{F} \) given by (7).

Substituting the robust policy into (24) 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 (8).

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.

### 39.6.2 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 (22), satisfies

...
CHAPTER 39. ROBUSTNESS

\[ J_F(x) = \max_{\psi \in \mathcal{P}} \left\{ x'(R + F'QF)x + \beta \left[ \int J_F((A - BF)x + Cw) \psi(dw) - \theta D_{KL}(\psi, \phi) \right] \right\} \]

Writing \( J_F(x) = x'P_Fx + d_F \) and applying the same argument used to derive (23) we get

\[ x'P_Fx + d_F = x'(R + F'QF)x + \beta \left[ x'(A - BF)' \mathcal{D}(P_F)(A - BF)x + d_F + \kappa(\theta, P_F) \right] \]

To solve this we take \( P_F \) to be the fixed point

\[ P_F = R + F'QF + \beta (A - BF)' \mathcal{D}(P_F)(A - BF) \]

and

\[ d_F := \frac{\beta}{1 - \beta} \kappa(\theta, P_F) = \frac{\beta}{1 - \beta} \theta \ln[\det(I - \theta^{-1}C'P_FC)^{-1}] \tag{27} \]

If you skip ahead to the appendix, you will be able to verify that \(-P_F\) is the solution to the Bellman equation in agent 2’s problem discussed above — we use this in our computations.

39.7 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

- \texttt{d\_operator()} and \texttt{b\_operator()} implement \( \mathcal{D} \) and \( \mathcal{B} \) respectively
- \texttt{robust\_rule()} and \texttt{robust\_rule\_simple()} both solve for the triple \( \hat{F}, \hat{K}, \hat{P} \), as described in equations (7) – (8) and the surrounding discussion
  - \texttt{robust\_rule()} is more efficient
  - \texttt{robust\_rule\_simple()} is more transparent and easier to follow
- \texttt{K\_to\_F()} and \texttt{F\_to\_K()} solve the decision problems of agent 1 and agent 2 respectively
- \texttt{compute\_deterministic\_entropy()} computes the left-hand side of (13)
- \texttt{evaluate\_F()} computes the loss and entropy associated with a given policy — see this discussion

39.8 Application

Let us consider a monopolist similar to this one, but now facing model uncertainty.

The inverse demand function is \( p_t = a_0 - a_1 y_t + d_t \)

where

\[ d_{t+1} = \rho d_t + \sigma_d w_{t+1}, \quad \{w_t\} \overset{iid}{\sim} N(0, 1) \]

and all parameters are strictly positive.
The period return function for the monopolist is

\[ r_t = p_t y_t - \gamma \frac{(y_{t+1} - y_t)^2}{2} - c y_t \]

Its objective is to maximize expected discounted profits, or, equivalently, to minimize \( \mathbb{E} \sum_{t=0}^{\infty} \beta^t (-r_t) \).

To form a linear regulator problem, we take the state and control to be

\[ x_t = \begin{bmatrix} 1 \\ y_t \\ d_t \end{bmatrix} \quad \text{and} \quad u_t = y_{t+1} - y_t \]

Setting \( b := (a_0 - c)/2 \) we define

\[
R = - \begin{bmatrix} 0 & b & 0 \\ b & -a_1 & 1/2 \\ 0 & 1/2 & 0 \end{bmatrix} \quad \text{and} \quad Q = \gamma/2
\]

For the transition matrices we set

\[ A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & \rho \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad C = \begin{bmatrix} 0 \\ 0 \\ \sigma_d \end{bmatrix} \]

Our aim is to compute the value-entropy correspondences shown above.

The parameters are

\[ a_0 = 100, \quad a_1 = 0.5, \quad \rho = 0.9, \quad \sigma_d = 0.05, \quad \beta = 0.95, \quad c = 2, \quad \gamma = 50.0 \]

The standard normal distribution for \( w_t \) is understood as the agent’s baseline, with uncertainty parameterized by \( \theta \).

We compute value-entropy correspondences for two policies.

1. The no concern for robustness policy \( F_0 \), which is the ordinary LQ loss minimizer.
2. A “moderate” concern for robustness policy \( F_b \), with \( \theta = 0.02 \).

The code for producing the graph shown above, with blue being for the robust policy, is as follows

In [3]: using QuantEcon, Plots, LinearAlgebra, Interpolations
gr(fmt = :png);

# model parameters
a_0 = 100
a_1 = 0.5
\( \rho = 0.9 \)
\( \sigma_d = 0.05 \)
\( \beta = 0.95 \)
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 .; 0.1  0.0 .; 0.0  0. ρ]
B = [0.1  0.0]'
C = [0.0  0. σ_d]'

## Functions

function evaluate_policy(θ, F)
    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(emax, F, bw, grid_size = 1000)
    if lowercase(bw) == "worst"
        θs = 1 ./ range(1e-8, 1000, length = grid_size)
    else
        θs = -1 ./ range(1e-8, 1000, length = grid_size)
    end
    data = zeros(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
end

## Main

# compute optimal rule
optimal_lq = QuantEcon.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 $\theta$ exceeds the breakdown point

```plaintext
test_matrix = I - (C' * Pb * C ./ \theta)[1]
eigenvals, eigenvecs = eigen(test_matrix)
```

```plaintext
@assert all(x -> x ≥ 0, eigenvals)
```

```plaintext
emax = 1.6e6
```

# compute values and entropies

```plaintext
optimal_best_case = value_and_entropy(emax, Fo, "best")
```

```plaintext
robust_best_case = value_and_entropy(emax, Fb, "best")
```

```plaintext
optimal_worst_case = value_and_entropy(emax, Fo, "worst")
```

```plaintext
robust_worst_case = value_and_entropy(emax, Fb, "worst")
```

# we reverse order of "worst_case"s so values are ascending

```plaintext
data_pairs = ((optimal_best_case, optimal_worst_case),
             (robust_best_case, robust_worst_case))
```

```plaintext
egrid = range(0, emax, length = 100)
egrid_data = []
for data_pair in data_pairs
    for data in data_pair
        x, y = data[:, 2], data[:, 1]
        curve = LinearInterpolation(x, y, extrapolation_bc = Line())
        push!(egrid_data, curve.(egrid))
    end
end
```

```plaintext
plot(egrid, egrid_data, color=[:
   red : red : blue : blue])
```

```plaintext
plot!(egrid, egrid_data[1], fillrange=egrid_data[2],
   fillcolor=:red, fillalpha=0.1, color=:red, legend=:none)
```

```plaintext
plot!(egrid, egrid_data[3], fillrange=egrid_data[4],
   fillcolor=:blue, fillalpha=0.1, color=:blue, legend=:none)
```

```plaintext
plot!(xlabel="Entropy", ylabel="Value")
```

Out[3]:

![Graph showing the relationship between entropy and value fields with shaded regions for optimal and robust cases.](image-url)
CHAPTER 39. ROBUSTNESS

Here’s another such figure, with \( \theta = 0.002 \) instead of 0.02

![Graph showing the value-entropy correspondence for the robust policy]

Can you explain the different shape of the value-entropy correspondence for the robust policy?

39.9 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 (8).

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 (7), then

\[
K(\hat{F}, \theta) = (\theta I - C'\hat{P}C)^{-1}C'\hat{P}(A - BF)
\]

**Proof:** As a first step, observe that when \( F = \hat{F} \), the Bellman equation associated with the LQ problem (11) – (12) is

\[
\hat{P} = -R - \hat{F}'Q\hat{F} - \beta^2 (A - BF)'\hat{P}C(\beta I + \beta C'\hat{P}C)^{-1}C'\hat{P}(A - BF) + \beta (A - BF)'\hat{P}(A - BF)
\]

(revisit this discussion if you don’t know where (29) comes from) and the optimal policy is

\[
w_{t+1} = -\beta(\beta I + \beta C'\hat{P}C)^{-1}C'\hat{P}(A - BF)x_t
\]

Suppose for a moment that \( -\hat{P} \) solves the Bellman equation (29).

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 (28).

Hence it remains only to show that \(-\hat{P}\) solves (29), or, in other words,

\[
\hat{P} = R + \hat{F}'Q\hat{F} + \beta(A - B \hat{F})' \hat{P}C(\theta I - C' \hat{P} C)^{-1} C' \hat{P}(A - B \hat{F}) + \beta(A - B \hat{F})' \hat{P}(A - B \hat{F})
\]

Using the definition of \(\mathcal{D}\), we can rewrite the right-hand side more simply as

\[
R + \hat{F}'Q\hat{F} + \beta(A - B \hat{F})' \mathcal{D}(\hat{P})(A - B \hat{F})
\]

Although it involves a substantial amount of algebra, it can be shown that the latter is just \(\hat{P}\).

(Hint: Use the fact that \(\hat{P} = \mathcal{B}(\mathcal{D}(\hat{P}))\))
Chapter 40

Discrete State Dynamic Programming

40.1 Contents

- Overview 40.2
- Discrete DPs 40.3
- Solving Discrete DPs 40.4
- Example: A Growth Model 40.5
- Exercises 40.6
- Solutions 40.7
- Appendix: Algorithms 40.8

40.2 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

40.2.1 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.

40.2.2 References

For background reading on dynamic programming and additional applications, see, for example,

- [68]
- [53], section 3.5
- [86]
- [100]
- [92]
- [78]
- EDTC, chapter 5

40.3 Discrete DPs

Loosely speaking, a discrete DP is a maximization problem with an objective function of the form

\[
\mathbb{E} \sum_{t=0}^{\infty} \beta^t r(s_t, a_t) \tag{1}
\]

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

### 40.3.1 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 [86] 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 (1).

Let’s now step through these ideas more carefully.

### 40.3.2 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 \( r: SA \rightarrow \mathbb{R} \)
4. A transition probability function \( Q: SA \rightarrow \Delta(S) \), where \( \Delta(S) \) is the set of probability distributions over \( S \)
5. A discount factor \( \beta \in [0, 1) \)

We also use the notation \( A := \bigcup_{s \in S} A(s) = \{0, \ldots, m - 1\} \) and call this set the action space. A policy is a function \( \sigma: S \rightarrow A \).
A policy is called \textit{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
\begin{itemize}
  \item the current reward at time \( t \) is \( r(s_t, \sigma(s_t)) \)
  \item the probability that \( s_{t+1} = s' \) is \( Q(s_t, \sigma(s_t), s') \)
\end{itemize}

For each \( \sigma \in \Sigma \), define
\begin{itemize}
  \item \( r_{\sigma} \) by \( r_{\sigma}(s) := r(s, \sigma(s)) \)
  \item \( Q_{\sigma} \) by \( Q_{\sigma}(s, s') := Q(s, \sigma(s), s') \)
\end{itemize}

Notice that \( Q_{\sigma} \) is a \textit{stochastic matrix} on \( S \).

It gives transition probabilities of the \textit{controlled chain} when we follow policy \( \sigma \).

If we think of \( r_{\sigma} \) as a column vector, then so is \( Q_{\sigma}^t r_{\sigma} \), and the \( s \)-th row of the latter has the interpretation
\[
(Q_{\sigma}^t r_{\sigma})(s) = \mathbb{E}[r(s_t, \sigma(s_t)) \mid s_0 = s] \quad \text{when} \quad \{s_t\} \sim Q_{\sigma}
\]  

\textbf{Comments}
\begin{itemize}
  \item \( \{s_t\} \sim Q_{\sigma} \) means that the state is generated by stochastic matrix \( Q_{\sigma} \)
  \item See this discussion on computing expectations of Markov chains for an explanation of the expression in (2)
\end{itemize}

Notice that we’re not really distinguishing between functions from \( S \) to \( \mathbb{R} \) and vectors in \( \mathbb{R}^n \).

This is natural because they are in one to one correspondence.

\subsection{40.3.3 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 (1) and use (2) to get
\[
v_{\sigma}(s) = \sum_{t=0}^{\infty} \beta^t (Q_{\sigma}^t r_{\sigma})(s) \quad (s \in S)
\]

This function is called the \textit{policy value function} for the policy \( \sigma \).

The \textit{optimal value function}, or simply \textit{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 \textit{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 \( v^*-\text{greedy}. \)

### 40.3.4 Two Operators

It is useful to define the following operators:

- The Bellman operator \( T : \mathbb{R}^S \rightarrow \mathbb{R}^S \) is defined by
  \[
  (Tv)(s) = \max_{a \in A(s)} \left\{ r(s, a) + \beta \sum_{s' \in S} v(s')Q(s, a, s') \right\} \quad (s \in S)
  \]
- For any policy function \( \sigma \in \Sigma \), the operator \( T_\sigma : \mathbb{R}^S \rightarrow \mathbb{R}^S \) is defined by
  \[
  (T_\sigma v)(s) = r(s, \sigma(s)) + \beta \sum_{s' \in S} v(s')Q(s, \sigma(s), s') \quad (s \in S)
  \]

This can be written more succinctly in operator notation as

\[
T_\sigma v = r_\sigma + \beta Q_\sigma v
\]

The two operators are both monotone

- \( v \leq w \) implies \( Tv \leq Tw \) pointwise on \( S \), and similarly for \( T_\sigma \)

They are also contraction mappings with modulus \( \beta \)

- \( \|Tv - Tw\| \leq \beta \|v - w\| \) and similarly for \( T_\sigma \), where \( \| \cdot \| \) is the max norm

For any policy \( \sigma \), its value \( v_\sigma \) is the unique fixed point of \( T_\sigma \).

For proofs of these results and those in the next section, see, for example, EDTC, chapter 10.

### 40.3.5 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

\[
\sigma^*(s) \in \arg\max_{a \in A(s)} \left\{ r(s, a) + \beta \sum_{s' \in S} v^*(s')Q(s, \sigma(s), s') \right\} \quad (s \in S)
\]

### 40.4 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.

### 40.4.1 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.

### 40.4.2 Policy Function Iteration

This routine, also known as Howard’s policy improvement algorithm, exploits more closely the particular structure of a discrete DP problem.

Each iteration consists of

1. A policy evaluation step that computes the value $v_\sigma$ of a policy $\sigma$ by solving the linear equation $v = T_\sigma v$.

2. A policy improvement step that computes a $v_\sigma$-greedy policy.

In the current setting policy iteration computes an exact optimal policy in finitely many iterations.

- See theorem 10.2.6 of EDTC for a proof

The details of the algorithm can be found in the appendix.

### 40.4.3 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.
40.5 Example: A Growth Model

Let’s consider a simple consumption-saving model.

A single household either consumes or stores its own output of a single consumption good.

The household starts each period with current stock $s$.

Next, the household chooses a quantity $a$ to store and consumes $c = s - a$

- Storage is limited by a global upper bound $M$
- Flow utility is $u(c) = c^a$

Output is drawn from a discrete uniform distribution on $\{0, ..., B\}$.

The next period stock is therefore

$$s' = a + U \quad \text{where} \quad U \sim U[0, ..., B]$$

The discount factor is $\beta \in [0, 1)$.

40.5.1 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 $S = \{0, ..., M + B\}$
  - hence $n = M + B + 1$
- the action to be the storage quantity $a$
- the set of feasible actions at $s$ to be $A(s) = \{0, ..., \min\{s, M\}\}$
  - hence $A = \{0, ..., M\}$ and $m = M + 1$
- the reward function to be $r(s, a) = u(s - a)$
- the transition probabilities to be

$$Q(s, a, s') := \begin{cases} \frac{1}{B+1} & \text{if } a \leq s' \leq a + B \\ 0 & \text{otherwise} \end{cases}$$

40.5.2 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).

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.

### 40.5.3 Setup

In [1]: using InstantiateFromURL
   # optionally add arguments to force installation: instantiate = true,
   # precompile = true
   github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

In [2]: using LinearAlgebra, Statistics, BenchmarkTools, Plots, QuantEcon
   using SparseArrays

In [3]: using BenchmarkTools, Plots, QuantEcon, Parameters
   gr(fmt = :png);

In [4]: SimpleOG = @with_kw (B = 10, M = 5, \( \alpha = 0.5 \), \( \beta = 0.9 \))

function transition_matrices(g)
   @unpack B, M, \( \alpha \), \( \beta \) = g
   u(c) = c^\( \alpha \)
   n = B + M + 1
   m = M + 1

   R = zeros(n, m)
   Q = zeros(n, m, n)

   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) : -\( \infty \))
      end
   end

   return (Q = Q, R = R)
end

Out[4]: transition_matrices (generic function with 1 method)

Let’s run this code and create an instance of SimpleOG

In [5]: g = SimpleOG();
   Q, R = transition_matrices(g);

   In case the preceding code was too concise, we can see a more verbose form

In [6]: function verbose_matrices(g)
   @unpack B, M, \( \alpha \), \( \beta \) = g
   u(c) = c^\( \alpha \)

   #Matrix dimensions. The +1 is due to the 0 state.
\[ n = B + M + 1 \]
\[ m = M + 1 \]

\[ R = \text{fill}(-\text{Inf}, n, m) \quad \# \text{Start assuming nothing is feasible} \]
\[ Q = \text{zeros}(n, m, n) \quad \# \text{Assume 0 by default} \]

#Create the R matrix

#Note: indexing into matrix complicated since Julia starts indexing at 1 instead of 0

#but the state s and choice a can be 0

for a in 0:M
    for s in 0:(B + M)
        if a <= s #i.e. if feasible
            R[s + 1, a + 1] = u(s - a)
        end
    end
end

#Create the Q multi-array

for s in 0:(B + M) #For each state
    for a in 0:M #For each action
        for sp in 0:(B + M) #For each state next period
            if( sp >= a && sp <= a + B) # The support of all realizations
                Q[s + 1, a + 1, sp + 1] = 1 / (B + 1) # Same prob of all
            end
        end
        @assert sum(Q[s + 1, a + 1, :]) ≈ 1 #Optional check that matrix is stochastic
    end
end

return (Q = Q, R = R)

Out[6]: \text{verbose_matrices (generic function with 1 method)}

Instances of \text{DiscreteDP} are created using the signature \text{DiscreteDP}(R, Q, \beta).

Let’s create an instance using the objects stored in g

In [7]: ddp = \text{DiscreteDP}(R, Q, g.β);

Now that we have an instance \text{ddp} of \text{DiscreteDP} we can solve it as follows

In [8]: results = solve(ddp, PFI)

Let’s see what we’ve got here

In [9]: fieldnames(typeof(results))

Out[9]: (:v, :Tv, :num_iter, :sigma, :mc)

The most important attributes are $v$, the value function, and $\sigma$, the optimal policy

In [10]: results.v

Out[10]: 16-element Array{Float64,1}:
  19.01740221695992
  20.017402216959916
  20.431615779333015
  20.749453024528794
  21.040780991093488
  21.30873018352461
  21.5447981610244
  21.769281810799864
  21.982703576083253
  22.1882432282385
  22.38450479651991
  22.578077363861723
  22.761091269771114
  23.11533958706524
  23.277617618874903
  23.43615779333015
  24.090909090909091

In [11]: results.sigma .- 1

Out[11]: 16-element Array{Int64,1}:
  0
  0
  0
  1
  1
  2
  2
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

In [12]: results.num_iter

Out[12]: 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

In [13]: stationary_distributions(results.mc)[1]

Out[13]: 16-element Array{Float64,1}:
  0.01732186732186732
  0.041210632119723034
  0.05773955773955773
  0.07426848335939244
  0.08095823095823096
  0.09090909090909091
  0.09090909090909093
  0.0909090909090909
  0.09090909090909091
  0.0909090909090909
  0.0909090909090909
  0.0735872235872236
  0.04968458789367884
  0.033169533169533166
  0.01664967549698462
  0.00950859950859951

Here’s the same information in a bar graph
What happens if the agent is more patient?

In [14]: g_2 = SimpleOG(β=0.99);
   Q_2, R_2 = transition_matrices(g_2);
   ddp_2 = DiscreteDP(R_2, Q_2, g_2.β)
   results_2 = solve(ddp_2, PFI)
   std_2 = stationary_distributions(results_2.mc)[1]

Out[14]: 16-element Array{Float64,1}:
          0.005469129800680602
          0.023213417598444343
          0.03147788040836169
          0.04800680602819641
          0.056271268838113765
          0.09090909090909091
          0.09090909090909093
          0.09090909090909093
          0.09090909090909094
          0.09090909090909093
          0.0854399611084103
          0.06769567331064659
          0.059431210500729234
          0.042902284880894495
          0.03463782207097716

In [15]: bar(std_2, label = "stationary dist")

Out[15]:
40.5. EXAMPLE: A GROWTH MODEL

We can see the rightward shift in probability mass.

40.5.4 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 $\text{DiscreteDP}(R, Q, \beta, s\_indices, a\_indices)$ where

- $s\_indices$ and $a\_indices$ are arrays of equal length $L$ enumerating all feasible state-action pairs
- $R$ is an array of length $L$ giving corresponding rewards
- $Q$ is an $L \times n$ transition probability array

Here’s how we could set up these objects for the preceding example

In [16]:

```python
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 = zeros(θ, n)
R = zeros(θ)
```
\[ b = 1 / (B + 1) \]

\[
\text{for } s \text{ in } 0:(M + B)
\]
\[
\text{for } a \text{ in } 0:\text{min}(M, s)
\]
\[
s\_indices = [s\_indices; s + 1]
\]
\[
a\_indices = [a\_indices; a + 1]
\]
\[
q = \text{zeros}(1, n)
\]
\[
q[(a + 1):((a + B) + 1)] .= b
\]
\[
Q = [Q; q]
\]
\[
R = [R; u(s-a)]
\]

\[
ddp = \text{DiscreteDP}(R, Q, \beta, s\_indices, a\_indices);
\]
\[
\text{results} = \text{solve}(ddp, \text{PFI})
\]

\textbf{Out[16]:} QuantEcon.DPSolveResult{PFI,Float64}([19.01740221695992, 20.
\text Slice omitted...
\]
\textbf{40.6 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 \texttt{DiscreteDP}.

\textbf{40.7 Solutions}

These were written jointly by Max Huber and Daisuke Oyama.
40.7.1 Setup

Details of the model can be found in the lecture. As in the lecture, we let \( f(k) = k^{0.65} \) with \( \alpha = 0.65 \), \( u(c) = \log c \), and \( \beta = 0.95 \).

In [17]:
\[
\begin{align*}
\alpha &= 0.65 \\
f(k) &= k^{\alpha} \\
u_{\log}(x) &= \log(x) \\
\beta &= 0.95
\end{align*}
\]

Out[17]: 0.95

Here we want to solve a finite state version of the continuous state model above. We discretize the state space into a grid of size \( \text{grid\_size} = 500 \), from \( 10^{-6} \) to \( \text{grid\_max}=2 \).

In [18]:
\[
\begin{align*}
\text{grid\_max} &= 2 \\
\text{grid\_size} &= 500 \\
\text{grid} &= \text{range}(1e-6, \text{grid\_max}, \text{length} = \text{grid\_size})
\end{align*}
\]

Out[18]:
\[
1.0e-6:0.004008014028056112:2.0
\]

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 \( 1, \ldots, \text{grid\_size} \). 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:

In [19]:
\[
\begin{align*}
\text{C} &= f.(\text{grid}) \\
\text{coord} &= \text{repeat}(\text{collect}(1:\text{grid\_size}), 1, \text{grid\_size}) \ #\text{coordinate matrix} \\
\text{s\_indices} &= \text{coord}[\text{C} > 0] \\
\text{a\_indices} &= \text{transpose}(\text{coord})[\text{C} > 0] \\
\text{L} &= \text{length}(\text{a\_indices})
\end{align*}
\]

Out[19]: 118841

Now let’s set up \( R \) and \( Q \).

In [20]:
\[
\text{R} = u_{\log}.(\text{C}[\text{C} > 0]);
\]
In [21]: using SparseArrays

    Q = spzeros(L, grid_size)  # Formerly spzeros

    for i in 1:L
        Q[i, a_indices[i]] = 1
    end

We’re now in a position to create an instance of DiscreteDP corresponding to the growth model.

In [22]: ddp = DiscreteDP(R, Q, β, s_indices, a_indices);

40.7.2 Solving the Model

In [23]: results = solve(ddp, PFI)
    v, α, num_iter = results.v, results.sigma, results.num_iter
    num_iter

Out[23]: 10

Let us compare the solution of the discrete model with the exact solution of the original continuous model. Here’s the exact solution:

In [24]: c = f(grid) - grid[σ]

    ab = α * β
    c1 = (log(1 - α * β) + log(α * β) * α * β / (1 - α * β)) / (1 - β)
    c2 = α / (1 - α * β)

    v_star(k) = c1 + c2 * log(k)
    c_star(k) = (1 - α * β) * k^α

Out[24]: c_star (generic function with 1 method)

Let’s plot the value functions.

In [25]: plot(grid, [v v_star(grid)], ylim = (-40, -32), lw = 2, label = ["discrete" "continuous"])

Out[25]: 
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.

```python
In [26]: plot(grid, [c c_star(grid)], lw = 2, label = ['discrete', 'continuous'],
children = legend = :topleft)
```

```
Out[26]:
```
These functions are again close, although some difference is visible and becomes more obvious as you zoom. Here are some statistics:

In [27]: maximum(abs(x - v_star(y)) for (x, y) in zip(v, grid))

Out[27]: 121.4981917053378

This is a big error, but most of the error occurs at the lowest gridpoint. Otherwise the fit is reasonable:

In [28]: maximum(abs(v[idx] - v_star(grid[idx])) for idx in 2:lastindex(v))

Out[28]: 0.012681735127500815

The value function is monotone, as expected:

In [29]: all(x - x ≥ 0, diff(v))

Out[29]: true

### 40.7.3 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.

In [30]: @benchmark results = solve(ddp, PFI)
   results = solve(ddp, PFI);

In [31]: @benchmark res1 = solve(ddp, VFI, max_iter = 500, epsilon = 1e-4)
   res1 = solve(ddp, VFI, max_iter = 500, epsilon = 1e-4);

In [32]: res1.num_iter

Out[32]: 294

In [33]: σ == res1.sigma

Out[33]: true

In [34]: @benchmark res2 = solve(ddp, MPFI, max_iter = 500, epsilon = 1e-4)
   res2 = solve(ddp, MPFI, max_iter = 500, epsilon = 1e-4);

In [35]: res2.num_iter

Out[35]: 16

In [36]: σ == res2.sigma

Out[36]: true
40.7.4 Replication of the figures

Let’s visualize convergence of value function iteration, as in the lecture.

```
In [37]: 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)
        push!(colors, RGBA(0, 0, 0, i/n))
    end

    plot(grid, w_init,
         ylims = (-40, -20),
         lw = 2,
         xlims = extrema(grid),
         label = "initial condition")

    plot!(grid, ws, label = "", color = reshape(colors, 1, length(colors)),
          lw = 2)
    plot!(grid, v_star.(grid), label = "true value function", color = :red,
          lw = 2)

Out[37]:
```

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.
In [38]: 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
    return c_policies
end

Out[38]: compute_policies (generic function with 1 method)

Now let’s generate the plots.

In [39]: 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"])
40.7.5 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 $k_0 = 0.1$.

In [40]:
\[
\text{discount_factors} = (0.9, 0.94, 0.98) \\
k_{\text{init}} = 0.1 \\
k_{\text{init\_ind}} = \text{findfirst}(\text{collect}(\text{grid}) \geq k_{\text{init}})
\]
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)))

Out[40]:

40.8 Appendix: Algorithms

This appendix covers the details of the solution algorithms implemented for DiscreteDP.

We will make use of the following notions of approximate optimality:

- For $\varepsilon > 0$, $v$ is called an $\varepsilon$-approximation of $v^*$ if $\|v - v^*\| < \varepsilon$
40.8. APPENDIX: ALGORITHMS

- A policy \( \sigma \in \Sigma \) is called \( \varepsilon \)-optimal if \( v_\sigma \) is an \( \varepsilon \)-approximation of \( v^* \)

40.8.1 Value Iteration

The **DiscreteDP** value iteration method implements value function iteration as follows:

1. Choose any \( v^0 \in \mathbb{R}^n \), and specify \( \varepsilon > 0 \); set \( i = 0 \).
2. Compute \( v^{i+1} = T v^i \).
3. If \( \|v^{i+1} - v^i\| < [(1 - \beta)/(2\beta)] \varepsilon \), then go to step 4; otherwise, set \( i = i + 1 \) and go to step 2.
4. Compute a \( v^{i+1} \)-greedy policy \( \sigma \), and return \( v^{i+1} \) and \( \sigma \).

Given \( \varepsilon > 0 \), the value iteration algorithm
- terminates in a finite number of iterations
- returns an \( \varepsilon/2 \)-approximation of the optimal value function and an \( \varepsilon \)-optimal policy function (unless \texttt{iter_max} is reached)

(While not explicit, in the actual implementation each algorithm is terminated if the number of iterations reaches \texttt{iter_max})

40.8.2 Policy Iteration

The **DiscreteDP** policy iteration method runs as follows:

1. Choose any \( v^0 \in \mathbb{R}^n \) and compute a \( v^0 \)-greedy policy \( \sigma^0 \); 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).

40.8.3 Modified Policy Iteration

The **DiscreteDP** modified policy iteration method runs as follows:

1. Choose any \( v^0 \in \mathbb{R}^n \) and specify \( \varepsilon > 0 \) and \( k \geq 0 \); set \( i = 0 \).
2. Compute a \( v^i \)-greedy policy \( \sigma^{i+1} \); 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 \( \text{span}(z) = \max(z) - \min(z) \)
1. 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.

2. Return \( v = u + [\beta/(1 - \beta)]([\min(u - v^i) + \max(u - v^i))/2]1 \) and \( \sigma_{i+1} \).

Given \( \varepsilon > 0 \), provided that \( v^0 \) is such that \( T v^0 \geq v^0 \), the modified policy iteration algorithm terminates in a finite number of iterations.

It returns an \( \varepsilon/2 \)-approximation of the optimal value function and an \( \varepsilon \)-optimal policy function (unless \( \text{iter\_max} \) is reached).

See also the documentation for \texttt{DiscreteDP}. 
Part V

Modeling in Continuous Time
Chapter 41

Modeling COVID 19 with Differential Equations

41.1 Contents

- Overview 41.2
- The SEIR Model 41.3
- Implementation 41.4
- Experiments 41.5
- Ending Lockdown 41.6

41.2 Overview

Coauthored with Chris Rackauckas

This is a Julia version of code for analyzing the COVID-19 pandemic.

The purpose of these notes is to introduce economists to quantitative modeling of infectious disease dynamics, and to modeling with ordinary differential equations.

In this lecture, dynamics are modeled using a standard SEIR (Susceptible-Exposed-Infected-Removed) model of disease spread, represented as a system of ordinary differential equations where the number of agents is large and there are no exogenous stochastic shocks.


We then extend this deterministic model in this lecture which build on this model, adding in aggregate shocks and policy tradeoffs.

The interest is primarily in

- studying the impact of suppression through social distancing on the spread of the infection
- the number of infections at a given time (which determines whether or not the health care system is overwhelmed); and
• how long the caseload can be deferred (hopefully until a vaccine arrives)

### 41.2.1 Setup

```julia
In [1]: using InstantiateFromURL
   # optionally add arguments to force installation: instantiate = true,
   # precompile = true
   github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")
```

```julia
In [2]: using LinearAlgebra, Statistics, Random, SparseArrays

In addition, we will be exploring the Ordinary Differential Equations package within the SciML ecosystem.
```

```julia
In [3]: using OrdinaryDiffEq
   using Parameters, Plots
   gr(fmt=:png);
```

### 41.3 The SEIR Model

In the version of the SEIR model, all individuals in the population are assumed to be in a finite number of states.

The states are: susceptible (S), exposed (E), infected (I) and removed (R).

This type of compartmentalized model has many extensions (e.g. SEIRS relaxes lifetime immunity and allow transitions from $R \rightarrow S$).

Comments:

• Those in state R have been infected and either recovered or died. Note that in some variations, R may refer only to recovered agents.
• Those who have recovered, and live, are assumed to have acquired immunity.
• Those in the exposed group are not yet infectious.

#### 41.3.1 Changes in the Infected State

Within the SEIR model, the flow across states follows the path $S \rightarrow E \rightarrow I \rightarrow R$.

We will ignore birth and non-covid death during our time horizon, and assume a large, constant, number of individuals of size $N$ throughout.

With this, the symbols $S, E, I, R$ are used for the total number of individuals in each state at each point in time, and $S(t) + E(t) + I(t) + R(t) = N$ for all $t$.

Since we have assumed that $N$ is large, we can use a continuum approximation for the number of individuals in each state.

The transitions between those states are governed by the following rates

• $\beta(t)$ is called the transmission rate or effective contact rate (the rate at which individuals bump into others and expose them to the virus).
• $\sigma$ is called the infection rate (the rate at which those who are exposed become infected)
• $\gamma$ is called the recovery rate (the rate at which infected people recover or die)
41.3. THE SEIR MODEL

The rate $\beta(t)$ is influenced by both the characteristics of the disease (e.g. the type and length of prolonged contact required for a transmission) and behavior of the individuals (e.g. social distancing, hygiene).

The SEIR model can then be written as

$$
\begin{align*}
\frac{dS}{dt} &= -\beta \frac{S}{N} I \\
\frac{dE}{dt} &= \beta \frac{S}{N} I - \sigma E \\
\frac{dI}{dt} &= \sigma E - \gamma I \\
\frac{dR}{dt} &= \gamma I
\end{align*}
$$

(1)

Here, $dy/dt$ represents the time derivative for the particular variable.

The first term of (1), $-\beta \frac{S}{N} I$, is the flow of individuals moving from $S \rightarrow E$, and highlights the underlying dynamics of the epidemic:

- Individuals in the susceptible state (S) have a rate $\beta(t)$ of prolonged contacts with other individuals where transmission would occur if either was infected.
- Of these contacts, a fraction $\frac{I(t)}{N}$ will be with infected agents (since we assumed that exposed individuals are not yet infectious).
- Finally, there are $S(t)$ susceptible individuals.
- The sign indicates that the product of those terms is the outflow from the $S$ state, and an inflow to the $E$ state.

### 41.3.2 Basic Reproduction Number

If $\beta$ was constant, then we could define $R_0 := \beta/\gamma$. This is the famous basic reproduction number for the SEIR model. See [51] for more details.

When the transmission rate is time-varying, we will follow notation in [31] and refer to $R_0(t)$ as a time-varying version of the basic reproduction number.

Analyzing the system in (1) provides some intuition on the $R_0(t) := \beta(t)/\gamma$ expression:

- Individual transitions from the infected to removed state occur at a Poisson rate $\gamma$, the expected time in the infected state is $1/\gamma$.
- Prolonged interactions occur at rate $\beta$, so a new individual entering the infected state will potentially transmit the virus to an average of $R_0 = \beta \times 1/\gamma$ others.
- In more complicated models, see [51] for a formal definition for arbitrary models, and an analysis on the role of $R_0 < 1$.

Note that the notation $R_0$ is standard in the epidemiology literature - though confusing, since $R_0$ is unrelated to $R$, the symbol that represents the removed state. For the remainder of the lecture, we will avoid using $R$ for removed state.

Prior to solving the model directly, we make a few changes to (1):

- Re-parameterize using $\beta(t) = \gamma R_0(t)$
- Define the proportion of individuals in each state as $s := S/N$ etc.
- Divide each equation in (1) by $N$, and write the system of ODEs in terms of the proportions.
\[
\begin{align*}
\frac{ds}{dt} &= -\gamma R_0 si \\
\frac{de}{dt} &= \gamma R_0 si - \sigma e \\
\frac{di}{dt} &= \sigma e - \gamma i \\
\frac{dr}{dt} &= \gamma i
\end{align*}
\] (2)

Since the states form a partition, we could reconstruct the “removed” fraction of the population as \(r = 1 - s - e - i\). However, keeping it in the system will make plotting more convenient.

### 41.3.3 Implementation

We begin by implementing a simple version of this model with a constant \(R_0\) and some baseline parameter values (which we discuss later).

First, define the system of equations

```
function F_simple(x, p, t; \gamma = 1/18, R\hat{0} = 3.0, \sigma = 1/5.2)
    s, e, i, r = x
    return [-\gamma*R\hat{0}*s*i; # ds/dt = -\gamma R\hat{0} si \\
            \gamma*R\hat{0}*s*i - \sigma*e; # de/dt = \gamma R\hat{0} si - \sigma e \\
            \sigma*e - \gamma*i; # di/dt = \sigma e - \gamma i \\
            \gamma*i; # dr/dt = \gamma i]
end
```

```
Out[4]: F_simple (generic function with 1 method)
```

Given this system, we choose an initial condition and a timespan, and create a `ODEProblem` encapsulating the system.

```
In [4]: i_0 = 1E-7
   # 33 = 1E-7 * 330 million population
~initially infected
   e_0 = 4.0 * i_0
   # 132 = 1E-7 * 330 million = initially exposed
   s_0 = 1.0 - i_0 - e_0
   r_0 = 0.0
   x_0 = [s_0, e_0, i_0, r_0] # initial condition
   tspan = (0.0, 350.0) # \approx 350 days
   prob = ODEProblem(F_simple, x_0, tspan)
```

```
Out[5]: ODEProblem with uType Array{Float64,1} and tType Float64. In-
   place: false
timespan: (0.0, 350.0)
u0: [0.9999995, 4.0e-7, 1.0e-7, 0.0]
```

With this, choose an ODE algorithm and solve the initial value problem. A good default algorithm for non-stiff ODEs of this sort might be `Tsit5()`, which is the Tsitouras 5/4 Runge-Kutta method).
In [6]: sol = solve(prob, Tsit5())
   plot(sol, labels = ["s" "e" "i" "r"], title = "SEIR Dynamics", lw = 2)

Out[6]:

We did not provide either a set of time steps or a *dt* time step size to the *solve*. Most accurate and high-performance ODE solvers appropriate for this problem use adaptive time-stepping, changing the step size based the degree of curvature in the derivatives.

Or, as an alternative visualization, the proportions in each state over time

In [7]: areaplot(sol.t, sol', labels = ["s" "e" "i" "r"], title = "SEIR\nProportions")

Out[7]:

While maintaining the core system of ODEs in \((s, e, i, r)\), we will extend the basic model to enable some policy experiments and calculations of aggregate values.

### 41.3.4 Extending the Model

First, we can consider some additional calculations such as the cumulative caseload (i.e., all those who have or have had the infection) as \(c = i + r\). Differentiating that expression and substituting from the time-derivatives of \(i(t)\) and \(r(t)\) yields \(\frac{dc}{dt} = \sigma e\).

We will assume that the transmission rate follows a process with a reversion to a value \(\bar{R}_0(t)\) which could conceivably be influenced by policy. The intuition is that even if the targeted \(\bar{R}_0(t)\) was changed through social distancing/etc., lags in behavior and implementation would smooth out the transition, where \(\eta\) governs the speed of \(R_0(t)\) moves towards \(\bar{R}_0(t)\).

\[
\frac{dR_0}{dt} = \eta(\bar{R}_0 - R_0) \tag{3}
\]

Finally, let \(\delta\) be the mortality rate, which we will leave constant. The cumulative deaths can be integrated through the flow \(\gamma i\) entering the “Removed” state.

Define the cumulative number of deaths as \(D(t)\) with the proportion \(d(t) := D(t)/N\).

\[
\frac{d}{dt}d(t) = \delta \gamma i \tag{4}
\]

While we could integrate the deaths given the solution to the model ex-post, it is more convenient to use the integrator built into the ODE solver. That is, we add \(\frac{d}{dt}d(t)\) rather than calculating \(d(t) = \int_0^t \delta \gamma i(\tau)d\tau\) ex-post.
This is a common trick when solving systems of ODEs. While equivalent in principle to using the appropriate quadrature scheme, this becomes especially convenient when adaptive time-stepping algorithms are used to solve the ODEs (i.e. there is not a regular time grid). Note that when doing so, \( d(0) = \int_0^0 \delta \gamma i(\tau) d\tau = 0 \) is the initial condition.

The system (2) and the supplemental equations can be written in vector form \( x := [\mathbf{s}, \mathbf{e}, \mathbf{i}, \mathbf{r}, R_0, \mathbf{c}, \mathbf{d}] \) with parameter tuple \( p := (\sigma, \gamma, \eta, \delta, \bar{R}_0(\cdot)) \)

Note that in those parameters, the targeted reproduction number, \( \bar{R}_0(t) \), is an exogenous function.

The model is then \( \frac{dx}{dt} = F(x, t) \) where,

\[
F(x, t) := \begin{bmatrix}
-\gamma R_0 s i \\
\gamma R_0 s i - \sigma e \\
\sigma e - \gamma i \\
\gamma i \\
\eta(\bar{R}_0(t) - R_0) \\
\sigma e \\
\delta \gamma i
\end{bmatrix}
\] \hspace{1cm} (5)

Note that if \( \bar{R}_0(t) \) is time-invariant, then \( F(x, t) \) is time-invariant as well.

### 41.3.5 Parameters

The parameters, \( \sigma, \delta, \) and \( \gamma \) should be thought of as parameters determined from biology and medical technology, and independent of social interactions.

As in Atkeson’s note, we set

- \( \sigma = 1/5.2 \) to reflect an average incubation period of 5.2 days.
- \( \gamma = 1/18 \) to match an average illness duration of 18 days.
- \( \bar{R}_0(t) = R_0(0) = 1.6 \) to match a **basic reproduction number** of 1.6, and initially time-invariant
- \( \delta = 0.01 \) for a one-percent mortality rate

As we will initially consider the case where \( R_0(0) = \bar{R}_0(0) \), the parameter \( \eta \) will not influence the first experiment.

### 41.4 Implementation

First, construct our \( F \) from (5)

```python
In [8]: def F(x, p, t):
    s, e, i, r, R0, c, d = x
    @unpack σ, γ, R0, η, δ = p
    return [-γ*R0*s*i; # ds/dt
            γ*R0*s*i - σ*e; # de/dt
            σ*e - γ*i; # di/dt
            γ*i; # dr/dt
            η*(R0(t, p) - R0); # dR0/dt
```
This function takes the vector $x$ of states in the system and extracts the fixed parameters passed into the $p$ object.

The only confusing part of the notation is the $R_n(t, p)$ which evaluates the $p \cdot R_n$ at this time (and also allows it to depend on the $p$ parameter).

### 41.4.1 Parameters

The baseline parameters are put into a named tuple generator (see previous lectures using Parameters.jl) with default values discussed above.

In [9]:

```julia
p_gen = @with_kw ( T = 550.0, γ = 1.0 / 18, σ = 1 / 5.2, η = 1.0 / 20, R₀⁻⁻ = 1.6, δ = 0.01, N = 3.3E8, R₀ = (t, p) -> p.R₀⁻⁻);
```

Note that the default $R₀(t)$ function always equals $R₀_n$ — a parameterizable natural level of $R₀$ used only by the $R₀⁻⁻$ function.

Setting initial conditions, we choose a fixed $s, i, e, r$, as well as $R₀(0) = R₀_n$ and $m(0) = 0.01$

In [10]: p = p_gen() # use all default parameters

```julia
i_0 = 1E-7
e_0 = 4.0 * i_0
s_0 = 1.0 - i_0 - e_0
x_0 = [s_0, e_0, i_0, 0.0, p.R₀⁻⁻, 0.0, 0.0]
tspan = (0.0, p.T)
prob = ODEProblem(F, x_0, tspan, p)
```

Out[10]: ODEProblem with uType Array{Float64,1} and tType Float64. In-place: false
timespan: (0.0, 550.0) u0: [0.99999995, 4.0e-7, 1.0e-7, 0.0, 1.6, 0.0, 0.0]

The $tspan$ of $(0.0, p.T)$ determines the $t$ used by the solver. The time scale needs to be consistent with the arrival rate of the transition probabilities (i.e. the $γ, σ$ were chosen based on daily data, so the unit of $t$ is a day).

The time period we investigate will be 550 days, or around 18 months:

### 41.5 Experiments

Let’s run some experiments using this code.
41.5. EXPERIMENTS

In [11]: sol = solve(prob, Tsit5())
    @show length(sol.t);

    length(sol.t) = 45

We see that the adaptive time-stepping used approximately 45 time-steps to solve this problem to the desired accuracy. Evaluating the solver at points outside of those time-steps uses an interpolator consistent with the solution to the ODE.

While it may seem that 45 time intervals is extremely small for that range, for much of the \( t \), the functions are very flat - and hence adaptive time-stepping algorithms can move quickly and interpolate accurately.

The solution object has built in plotting support.

In [12]: plot(sol, vars = [6, 7], label = ["c(t)" "d(t)"], lw = 2,
        title = ["Cumulative Infected" "Death Proportion"],
        layout = (1,2), size = (900, 300))

Out[12]:

A few more comments:

- If you want to ensure that there are specific points that the adaptive-time stepping must include (e.g. at known discontinuities) use \texttt{tstops}.
- The built-in plots for the solutions provide all of the attributes in \texttt{Plots.jl}.
- See here for details on analyzing the solution and extracting the output.

41.5.1 Experiment 1: Constant Reproduction Case

Let’s start with the case where \( \bar{R}_0(t) = R_{0n} \) is constant.

We calculate the time path of infected people under different assumptions of \( R_{0n} \):

In [13]: R̄_n_vals = range(1.6, 3.0, length = 6)
    sols = [solve(ODEProblem(F, x_0, tspan, p_gen(R̄_n = R̄_n)),
                  Tsit5(), saveat=0.5) for R̄_n in R̄_n_vals];

Here we chose \texttt{saveat=0.5} to get solutions that were evenly spaced every 0.5.
Changing the saved points is just a question of storage/interpolation, and does not change
the adaptive time-stepping of the solvers.

Let’s plot current cases as a fraction of the population.

In [14]:
labels = permutedims(["R_0 = $r" for r in R0_n_vals])
infecteds = [sol[3,:] for sol in sols]
plot(infecteds, label=labels, legend=:topleft, lw = 2, xlabel = "t",
     ylabel = "i(t)", title = "Current Cases")

Out[14]:

As expected, lower effective transmission rates defer the peak of infections.
They also lead to a lower peak in current cases.

Here is cumulative cases, as a fraction of population:

In [15]:
cumulative_infected = [sol[6,:] for sol in sols]
plot(cumulative_infected, label=labels, legend=:topleft, lw = 2, xlabel = "t",
     ylabel = "c(t)", title = "Cumulative Cases")

Out[15]:
41.5.2 Experiment 2: Changing Mitigation

Let’s look at a scenario where mitigation (e.g., social distancing) is successively imposed, but the target (maintaining $R_{0n}$) is fixed.

To do this, we start with $R_0(0) \neq R_{0n}$ and examine the dynamics using the $\frac{dR_0}{dt} = \eta(R_{0n} - R_0)$ ODE.

In the simple case, where $\bar{R}_0(t) = R_{0n}$ is independent of the state, the solution to the ODE given an initial condition is $R_0(t) = R_0(0)e^{-\eta t} + R_{0n}(1 - e^{-\eta t})$

We will examine the case where $R_0(0) = 3$ and then it falls to $R_{0n} = 1.6$ due to the progressive adoption of stricter mitigation measures.

The parameter $\eta$ controls the rate, or the speed at which restrictions are imposed.

We consider several different rates:

In [16]: $\eta_{vals} = [1/5, 1/10, 1/20, 1/50, 1/100]$
labels = permutedims(["eta = $\eta$ for $\eta$ in $\eta_{vals}$]);

Let’s calculate the time path of infected people, current cases, and mortality

In [17]: $x_\theta = [s_\theta, e_\theta, i_\theta, 0, 0, 0, 0, 0, 0, 0, 0, 0]$
sols = [solve(ODEProblem(F, x_\theta, tsp, p_gen(\eta=\eta)), Tsit5(), saveat=0.5)]$

for $\eta$ in $\eta_{vals}$;

Next, plot the $R_0$ over time:

In [18]: $Rs = [sol[5,:] for sol in sols]$
plot(Rs, label=labels, legend=:topright, lw = 2, xlabel = "t",
ylabel = "R_0(t)", title = "Basic Reproduction Number")
Out[18]:

```
In [18]:
```

![Graph showing the basic reproduction number over time.]

Now let's plot the number of infected persons and the cumulative number of infected persons:

```
In [19]: infecteds = [sol[3,:] for sol in sols]
plot(infecteds, label=labels, legend=topleft, lw=2, xlabel="t",
ylabel="i(t)", title="Current Infected")
```

Out[19]:

```
In [19]:
```
41.6 Ending Lockdown

The following is inspired by additional results by Andrew Atkeson on the timing of lifting lockdown.

Consider these two mitigation scenarios:

1. choose $\bar{R}_0(t)$ to target $R_0(t) = 0.5$ for 30 days and then $R_0(t) = 2$ for the remaining 17 months. This corresponds to lifting lockdown in 30 days.

2. $R_0(t) = 0.5$ for 120 days and then $R_0(t) = 2$ for the remaining 14 months. This corresponds to lifting lockdown in 4 months.

For both of these, we will choose a large $\eta$ to focus on the case where rapid changes in the lockdown policy remain feasible.

The parameters considered here start the model with 25,000 active infections and 75,000 agents already exposed to the virus and thus soon to be contagious.
\[
p_{\text{early}} = p_{\text{gen}}(R_0 = R_0^{\text{lift\_early}}, \eta = 10.0) \\
p_{\text{late}} = p_{\text{gen}}(R_0 = R_0^{\text{lift\_late}}, \eta = 10.0)
\]

\[
\begin{align*}
# initial conditions \\
i_0 &= 25000 / p_{\text{early}}.N \\
e_0 &= 75000 / p_{\text{early}}.N \\
s_0 &= 1.0 - i_0 - e_0 \\
x_0 &= [s_0, e_0, i_0, 0.0, R_L, 0.0, 0.0] \# start in lockdown
\end{align*}
\]

\[
\begin{align*}
# create two problems, with rapid movement of R_\theta(t) towards R_\theta(t) \\
prob_{\text{early}} &= \text{ODEProblem}(F, x_0, tspan, p_{\text{early}}) \\
prob_{\text{late}} &= \text{ODEProblem}(F, x_0, tspan, p_{\text{late}})
\end{align*}
\]

Out[21]: ODEProblem with uType Array{Float64,1} and tType Float64. In-place: false 
timespan: (0.0, 550.0) 
u0: [0.9996969696969696, 0.00022727272727272727, 7.575757575757576e-5, 0.
\]
\]

Unlike the previous examples, the $R_\theta(t)$ functions have discontinuities which might occur. We can improve the efficiency of the adaptive time-stepping methods by telling them to include a step exactly at those points by using \texttt{tstops}

Let's calculate the paths:

In [22]: sol_early = solve(prob_early, Tsit5(), tstops = [30.0, 120.0]) 
sol_late = solve(prob_late, Tsit5(), tstops = [30.0, 120.0]) 
plot(sol_early, vars = [7], title = "Total Mortality", label = "Lift Early") 
plot!(sol_late, vars = [7], label = "Lift Late")

Out[22]:
Next we examine the daily deaths, \( \frac{dD(t)}{dt} = N \delta \gamma i(t) \).

In [23]: flow_deaths(sol, p) = p.N * p.δ * p.γ * sol[3,:]

plot(sol_early.t, flow_deaths(sol_early, p_early), title = "Flow Deaths",
     label = "Lift Early")
plot!(sol_late.t, flow_deaths(sol_late, p_late), label = "Lift Late")

Out[23]:

![Flow Deaths Graph](image-url)
Pushing the peak of curve further into the future may reduce cumulative deaths if a vaccine is found, or allow health authorities to better smooth the caseload.

### 41.6.1 Randomness

Despite its richness, the model above is fully deterministic. The policy $\bar{R}_0(t)$ could change over time, but only in predictable ways.

One way that randomness can lead to aggregate fluctuations is the granularity that comes through the discreteness of individuals. This topic, the connection between SDEs and the Langevin equations typically used in the approximation of chemical reactions in well-mixed media is explored in further lectures on continuous time Markov chains.

Instead, in the next lecture, we will concentrate on randomness that comes from aggregate changes in behavior or policy.
Chapter 42

Modeling Shocks in COVID 19 with Stochastic Differential Equations

42.1 Contents

- Overview 42.2
- The Basic SIR/SIRD Model ??
- Introduction to SDEs 42.4
- Ending Lockdown 42.5
- Reinfection 42.6

42.2 Overview

Coauthored with Chris Rackauckas

This lecture continues the analyzing of the COVID-19 pandemic established in this lecture.


Here we extend the model to include policy-relevant aggregate shocks.

42.2.1 Continuous-Time Stochastic Processes

In continuous-time, there is an important distinction between randomness that leads to continuous paths vs. those which have (almost surely right-continuous) jumps in their paths. The most tractable of these includes the theory of Levy Processes.

Among the appealing features of Levy Processes is that they fit well into the sorts of Markov modeling techniques that economists tend to use in discrete time, and usually fulfill the measurability required for calculating expected present discounted values.

Unlike in discrete-time, where a modeler has license to be creative, the rules of continuous-time stochastic processes are much stricter. One can show that a Levy process’s noise can be decomposed into two portions:
1. **Weiner Processes** (as known as Brownian Motion) which leads to a diffusion equations, and is the only continuous-time Levy process with continuous paths.

2. **Poisson Processes** with an arrival rate of jumps in the variable.

Every other Levy Process can be represented by these building blocks (e.g. a **Diffusion Process** such as Geometric Brownian Motion is a transformation of a Weiner process, a **jump diffusion** is a diffusion process with a Poisson arrival of jumps, and a continuous-time markov chain (CMTC) is a Poisson process jumping between a finite number of states).

In this lecture, we will examine shocks driven by transformations of Brownian motion, as the prototypical Stochastic Differential Equation (SDE).

### 42.2.2 Setup

```julia
In [1]: using InstantiateFromURL
   # optionally add arguments to force installation: instantiate = true, precompile = true
github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")
```

```julia
In [2]: using LinearAlgebra, Statistics, Random, SparseArrays
```

In addition, we will be exploring packages within the **SciML ecosystem** and others covered in previous lectures.

```julia
In [3]: using OrdinaryDiffEq, StochasticDiffEq
   using Parameters, Plots
   gr(fmt=:png);
```

### 42.3 The Basic SIR/SIRD Model

To demonstrate another common compartmentalized model we will change the previous SEIR model to remove the exposed state, and more carefully manage the death state, D.

The states are are now: susceptible (S), infected (I), resistant (R), or dead (D).

Comments:

- Unlike the previous SEIR model, the R state is only for those recovered, alive, and currently resistant.
- As before, we start by assuming those have recovered have acquired immunity.
- Later, we could consider transitions from R to S if resistance is not permanent due to virus mutation, etc.

#### 42.3.1 Transition Rates

See the previous lecture, for a more detailed development of the model.

- $\beta(t)$ is called the *transmission rate* or *effective contact rate* (the rate at which individuals bump into others and expose them to the virus).
- $\gamma$ is called the *resolution rate* (the rate at which infected people recover or die).
- $\delta(t) \in [0, 1]$ is the *death probability*.
42.4. INTRODUCTION TO SDES

As before, we re-parameterize as \( R_0(t) := \beta(t)/\gamma \), where \( R_0 \) has previous interpretation
Jumping directly to the equations in \( s, i, r, d \) already normalized by \( N \),

\[
\begin{align*}
    ds &= -\gamma R_0 s i dt \\
    di &= (\gamma R_0 s i - \gamma i) dt \\
    dr &= (1 - \delta)\gamma i dt \\
    dd &= \delta \gamma i dt
\end{align*}
\]

(1)

Note that the notation has changed to heuristically put the \( dt \) on the right hand side, which will be used when adding the stochastic shocks.

42.4 Introduction to SDEs

We start by extending our model to include randomness in \( R_0(t) \) and then the mortality rate \( \delta(t) \).
The result is a system of Stochastic Differential Equations (SDEs).

42.4.1 Shocks to Transmission Rates

As before, we assume that the basic reproduction number, \( R_0(t) \), follows a process with a reversion to a value \( \bar{R}_0(t) \) which could conceivably be influenced by policy. The intuition is that even if the targeted \( \bar{R}_0(t) \) was changed through social distancing/etc., lags in behavior and implementation would smooth out the transition, where \( \eta \) governs the speed of \( R_0(t) \) moves towards \( \bar{R}_0(t) \).

Beyond changes in policy, randomness in \( R_0(t) \) may come from shocks to the \( \beta(t) \) process.
For example,

- Misinformation on Facebook spreading non-uniformly.
- Large political rallies, elections, or protests.
- Deviations in the implementation and timing of lockdown policy between demographics, locations, or businesses within the system.
- Aggregate shocks in opening/closing industries.

To implement these sorts of randomness, we will add on a diffusion term with an instantaneous volatility of \( \sigma \sqrt{R_0} \).

- This equation is used in the Cox-Ingersoll-Ross and Heston models of interest rates and stochastic volatility.
- The scaling by the \( \sqrt{R_0} \) ensure that the process stays weakly positive. The heuristic explanation is that the variance of the shocks converges to zero as \( R_0 \) goes to zero, enabling the upwards drift to dominate.
- See here for a heuristic description of when the process is weakly and strictly positive.

The notation for this SDE is then

\[
dR_{0t} = \eta(\bar{R}_{0t} - R_{0t})dt + \sigma \sqrt{R_{0t}}dW_t
\]

(2)

where \( W \) is standard Brownian motion (i.e a Weiner Process).
Heuristically, if \( \sigma = 0 \), divide this equation by \( dt \) and it nests the original ODE used in the previous lecture.

While we do not consider any calibration for the \( \sigma \) parameter, empirical studies such as *Estimating and Simulating a SIRD Model of COVID-19 for Many Countries, States, and Cities* (Figure 6) show highly volatile \( R_0(t) \) estimates over time.

Even after lockdowns are first implemented, we see variation between 0.5 and 1.5. Since countries are made of interconnecting cities with such variable contact rates, a high \( \sigma \) seems reasonable both intuitively and empirically.

### 42.4.2 Mortality Rates

Unlike the previous lecture, we will build up towards mortality rates which change over time.

Imperfect mixing of different demographic groups could lead to aggregate shocks in mortality (e.g. if a retirement home is afflicted vs. an elementary school). These sorts of relatively small changes might be best modeled as a continuous path process.

Let \( \delta(t) \) be the mortality rate and in addition,

- Assume that the base mortality rate is \( \bar{\delta} \), which acts as the mean of the process, reverting at rate \( \theta \). In more elaborate models, this could be time-varying.
- The diffusion term has a volatility \( \xi \sqrt{\delta(1-\delta)} \).
- As the process gets closer to either \( \delta = 1 \) or \( \delta = 0 \), the volatility goes to 0, which acts as a force to allow the mean reversion to keep the process within the bounds
- Unlike the well-studied Cox-Ingersoll-Ross model, we make no claims on the long-run behavior of this process, but will be examining the behavior on a small timescale so this is not an issue.

Given this, the stochastic process for the mortality rate is,

\[
d\delta_t = \theta(\bar{\delta} - \delta_t)dt + \xi \sqrt{\delta_t(1-\delta_t)}dW_t
\]

(3)

Where the \( W_t \) Brownian motion is independent from the previous process.

### 42.4.3 System of SDEs

The system (1) can be written in vector form \( x := [s, i, r, d, R_0, \delta] \) with parameter tuple parameter tuple \( p := (\gamma, \eta, \sigma, \theta, \xi, \bar{R}_0(\cdot), \bar{\delta}) \)

The general form of the SDE is.

\[
dx_t = F(x_t, t; p)dt + G(x_t, t; p)dW
\]

With the drift,

\[
F(x, t; p) := \begin{bmatrix}
-\gamma R_0 s i \\
\gamma R_0 s i - \gamma i \\
(1-\delta)\gamma i \\
\delta \gamma i \\
\eta(\bar{R}_0(t) - R_0) \\
\theta (\delta - \delta)
\end{bmatrix}
\]

(4)
Here, it is convenient but not necessary for \( dW \) to have the same dimension as \( x \). If so, then we can use a square matrix \( G(x,t;p) \) to associate the shocks with the appropriate \( x \) (e.g. diagonal noise, or using a covariance matrix).

As the two independent sources of Brownian motion only affect the \( dR_0 \) and \( d\delta \) terms (i.e. the 5th and 6th equations), define the covariance matrix as

\[
G(x,t) := \text{diag} \left( \begin{array}{cccc}
0 & 0 & 0 & \sigma \sqrt{R_0} \\
0 & 0 & 0 & \xi \sqrt{\delta(1 - \delta)}
\end{array} \right)
\] (5)

42.4.4 Implementation

First, construct our \( F \) from (4) and \( G \) from (5)

In [4]: `function F(x, p, t)`
   `s, i, r, d, Ṙ, δ = x`
   `@unpack γ, Ṙ₀, η, σ, ξ, θ, δ_bar = p`
   `return [-γ*R₀*s*i;  # ds/dt`
             γ*R₀*s*i - γ*i;  # di/dt`
             (1-δ)*γ*i;       # dr/dt`
             δ*γ*i;          # dd/dt`
             η*(Ṙ₀(t, p) - Ṙ₀); # dṘ₀/dt`
             θ*(δ_bar - δ);   # dδ/dt`]
`end`

`function G(x, p, t)`
   `s, i, r, d, Ṙ, δ = x`
   `@unpack γ, Ṙ₀, η, σ, ξ, θ, δ_bar = p`
   `return [0; 0; 0; 0; σ*sqrt(Ṙ₀); ξ*sqrt(δ*(1-δ))]`
`end`

Out[4]: G (generic function with 1 method)

Next create a settings generator, and then define a SDEProblem with Diagonal Noise.

In [5]: `p_gen = @with_kw (T = 550.0, γ = 1.0 / 18, η = 1.0 / 20,
                       Ṙ₀ₙ = 1.6, Ṙ₀ = (t, p) -> p.Ṙ₀ₙ, δ_bar = 0.01,
                       σ = 0.03, ξ = 0.004, θ = 0.2, N = 3.3E8)`
   `p = p_gen() # use all defaults`
   `i₀ = 25000 / p.N`
   `r₀ = 0.0`
   `d₀ = 0.0`
   `s₀ = 1.0 - i₀ - r₀ - d₀`
   `Ṙ₀₀ = 0.5 # starting in lockdown`
   `δ₀ = p.δ_bar`
   `x₀ = [s₀, i₀, r₀, d₀, Ṙ₀₀, δ₀]`
   `prob = SDEProblem(F, G, x₀, (θ, p.T), p)`
Out[5]: SDEProblem with uType Array{Float64,1} and tType Float64. In-
place: false
timespan: (0.0, 550.0)
u0: [0.9999242424242424, 7.575757575757576e-5, 0.0, 0.0, 0.5, 0.01]

We solve the problem with the SOSRI algorithm (Adaptive strong order 1.5 methods for diagonal noise Ito and Stratonovich SDEs).

In [6]: sol_1 = solve(prob, SOSRI());
   @show length(sol_1.t);

   length(sol_1.t) = 628

As in the deterministic case of the previous lecture, we are using an adaptive time-stepping method. However, since this is an SDE, (1) you will tend to see more timesteps required due to the greater curvature; and (2) the number of timesteps will change with different shock realizations.

With stochastic differential equations, a “solution” is akin to a simulation for a particular realization of the noise process.

If we take two solutions and plot the number of infections, we will see differences over time:

In [7]: sol_2 = solve(prob, SOSRI())
plot(sol_1, vars=[2], title = "Number of Infections", label = "Trajectory1"
     lm = 2, xlabel = "t", ylabel = "i(t)"
plot!(sol_2, vars=[2], label = "Trajectory 2", lm = 2, ylabel = "i(t)"

Out[7]:

![Number of Infections](image)
The same holds for other variables such as the cumulative deaths, mortality, and $R_0$:

\textbf{In [8]}: plot\_1 = plot(sol\_1, vars=[4], title = "Cumulative Death Proportion", 𝑑label = "Trajectory 1",

lw = 2, xlabel = "t", ylabel = "d(t)", legend = :topleft)

plot\_2 = plot(sol\_1, vars=[3], title = "Cumulative Recovered Proportion", 𝑑label = "Trajectory 1",

lw = 2, xlabel = "t", ylabel = "d(t)", legend = :topleft)

plot\_3 = plot(sol\_1, vars=[5], title = "R_0 transition from lockdown", 𝑑label = "Trajectory 1",

lw = 2, xlabel = "t", ylabel = "R_0(t)", ylim = (0.006, 0.014))

plot\_4 = plot(sol\_1, vars=[6], title = "Mortality Rate", label = "Trajectory 1",

lw = 2, xlabel = "t", ylabel = "delta(t)", ylim = (0.006, 0.014))

plot(plot\_1, plot\_2, plot\_3, plot\_4, size = (900, 600))

\textbf{Out[8]}:

See here for comments on finding the appropriate SDE algorithm given the structure of $F(x, t)$ and $G(x, t)$

- If $G$ has diagonal noise (i.e. $G(x, t)$ is a diagonal, and possibly a function of the state), then \textbf{SOSRI} is the typical choice.
• If $G$ has additive (i.e. $G(t)$ is a independent from the state), then SOSRA is usually the best algorithm for even mildly stiff $F$.
• If the noise process is more general, LambdaEM and RKMilGeneral are flexible to all noise processes.
• If high accuracy and adaptivity are not required, then EM (i.e. Euler-Maruyama method typically used by economists) is flexible in its ability to handle different noise processes.

### 42.4.5 Ensembles

While individual simulations are useful, you often want to look at an ensemble of trajectories of the SDE in order to get an accurate picture of how the system evolves.

To do this, use the EnsembleProblem in order to have the solution compute multiple trajectories at once. The returned EnsembleSolution acts like an array of solutions but is imbued to plot recipes to showcase aggregate quantities.

For example:

**In [9]:**
```
ensembleprob = EnsembleProblem(prob)
sol = solve(ensembleprob, SOSRI(), EnsembleSerial(), trajectories = 10)
plot(sol, vars = [2], title = "Infection Simulations", ylabel = "i(t)", xlabel = "t", lm = 2)
```

**Out[9]:**

![Infection Simulations](attachment:infection_simulations.png)

Or, more frequently, you may want to run many trajectories and plot quantiles, which can be automatically run in parallel using multiple threads, processes, or GPUs. Here we showcase EnsembleSummary which calculates summary information from an ensemble and plots the mean of the solution along with calculated quantiles of the simulation:
In [10]: trajectories = 100  # choose larger for smoother quantiles
    sol = solve(ensembleprob, SOSRI(), EnsembleThreads(), trajectories = trajectories)
    summ = EnsembleSummary(sol)  # defaults to saving 0.05, 0.95 quantiles
    plot(summ, idxs = (2,), title = "Quantiles of Infections Ensemble", ylabel = "i(t)",
         xlabel = "t", labels = "Middle 95% Quantile", legend = :topright)

Out[10]:

In addition, you can calculate more quantiles and stack graphs

In [11]: sol = solve(ensembleprob, SOSRI(), EnsembleThreads(), trajectories = trajectories)
    summ = EnsembleSummary(sol)  # defaults to saving 0.05, 0.95 quantiles
    summ2 = EnsembleSummary(sol, quantiles = (0.25, 0.75))

    plot(summ, idxs = (2,4,5,6),
         title = ["Proportion Infected", "Proportion Dead", "R_0", "delta"],
         ylabel = ["i(t)", "d(t)", "R_0(t)", "delta(t)"], xlabel = "t",
         legend = [:topleft :topleft :bottomright :bottomright],
         labels = "Middle 95% Quantile", layout = (2,2), size = (900, 600))
    plot!(summ2, idxs = (2,4,5,6),
          labels = "Middle 50% Quantile", legend = [:topleft :topleft :bottomright :bottomright])

Out[11]:

In [11]: trajectories = 100  # choose larger for smoother quantiles
    sol = solve(ensembleprob, SOSRI(), EnsembleThreads(), trajectories = trajectories)
    summ = EnsembleSummary(sol)  # defaults to saving 0.05, 0.95 quantiles
    plot(summ, idxs = (2,), title = "Quantiles of Infections Ensemble", ylabel = "i(t)",
         xlabel = "t", labels = "Middle 95% Quantile", legend = :topright)

Out[10]:

In addition, you can calculate more quantiles and stack graphs

In [11]: sol = solve(ensembleprob, SOSRI(), EnsembleThreads(), trajectories = trajectories)
    summ = EnsembleSummary(sol)  # defaults to saving 0.05, 0.95 quantiles
    summ2 = EnsembleSummary(sol, quantiles = (0.25, 0.75))

    plot(summ, idxs = (2,4,5,6),
         title = ["Proportion Infected", "Proportion Dead", "R_0", "delta"],
         ylabel = ["i(t)", "d(t)", "R_0(t)", "delta(t)"], xlabel = "t",
         legend = [:topleft :topleft :bottomright :bottomright],
         labels = "Middle 95% Quantile", layout = (2,2), size = (900, 600))
    plot!(summ2, idxs = (2,4,5,6),
          labels = "Middle 50% Quantile", legend = [:topleft :topleft :bottomright :bottomright])

Out[11]:

In addition, you can calculate more quantiles and stack graphs
Some additional features of the ensemble and SDE infrastructure are

- Plotting
- Noise Processes, Non-diagonal noise, and Correlated Noise
- Parallel Ensemble features
- Transforming the ensemble calculations with an output_func or reduction
- Auto-GPU accelerated by using EnsembleGPUArray() from DiffEqGPU

### 42.4.6 Changing Mitigation

Consider a policy maker who wants to consider the impact of relaxing lockdown at various speeds.

We will shut down the shocks to the mortality rate (i.e. $\xi = 0$) to focus on the variation caused by the volatility in $R_0(t)$.

Consider $\eta = 1/50$ and $\eta = 1/20$, where we start at the same initial condition of $R_0(0) = 0.5$.

In [12]: function generate_\eta_experiment(\eta; p_gen = p_gen, trajectories = 100, saveat = 1.0, x_0 = x_0, T = 120.0)
   p = p_gen(\eta = \eta, \xi = 0.0)
   ensembleprob = EnsembleProblem(SDEProblem(F, G, x_0, (0, T), p))
   sol = solve(ensembleprob, SOSRI(), EnsembleThreads(), trajectories = trajectories, saveat = saveat)
   return EnsembleSummary(sol)
end

# Evaluate two different lockdown scenarios
\eta_1 = 1/50
\eta_2 = 1/20
summ_1 = generate_\eta_experiment(\eta_1)
summ_2 = generate_\eta_experiment(\eta_2)
While the mean of the $d(t)$ increases, unsurprisingly, we see that the 95\% quantile for later time periods is also much larger - even after the $R_0$ has converged.

That is, volatile contact rates (and hence $R_0$) can interact to make catastrophic worst-case scenarios due to the nonlinear dynamics of the system.
42.5 Ending Lockdown

As in the deterministic lecture, we can consider two mitigation scenarios

1. choose $\bar{R}_0(t)$ to target $R_0 = 0.5$ for 30 days and then $R_0 = 2$ for the remaining 17 months. This corresponds to lifting lockdown in 30 days.

2. target $R_0 = 0.5$ for 120 days and then $R_0 = 2$ for the remaining 14 months. This corresponds to lifting lockdown in 4 months.

Since empirical estimates of $R_0(t)$ discussed in [31] and other papers show it to have wide variation, we will maintain a fairly larger $\sigma$.

We start the model with 100,000 active infections.

In [13]:

```
R_0_L = 0.5  # lockdown
η_experiment = 1.0/10
σ_experiment = 0.04
R_lift_early(t, p) = t < 30.0 ? R_0_L : 2.0
R_lift_late(t, p) = t < 120.0 ? R_0_L : 2.0

p_early = p_gen(R = R_lift_early, η = η_experiment, σ = σ_experiment)
p_late = p_gen(R = R_lift_late, η = η_experiment, σ = σ_experiment)

# initial conditions
i_0 = 100000 / p_early.N
r_0 = 0.0
d_0 = 0.0
s_0 = 1.0 - i_0 - r_0 - d_0
δ_0 = p_early.δ_bar

x_0 = [s_0, i_0, r_0, d_0, R_0_L, δ_0]  # start in lockdown
prob_early = SDEProblem(F, G, x_0, (0, p_early.T), p_early)
prob_late = SDEProblem(F, G, x_0, (0, p_late.T), p_late)
```

Out[13]:
SDEProblem with uType Array{Float64,1} and tType Float64. In-place: false

timespan: (0.0, 550.0)
u0: [0.9996969696969698, 0.000303030303030303, 0.0, 0.0, 0.5, 0.01]

Simulating for a single realization of the shocks, we see the results are qualitatively similar to what we had before

In [14]:

```
sol_early = solve(prob_early, SOSRI())
sol_late = solve(prob_late, SOSRI())
```

plot(sol_early, vars = [5, 1, 2, 4],
title = ["R_0" "Susceptible" "Infected" "Dead"],
layout = (2, 2), size = (900, 600),
ylabel = ["R_0(t)" "s(t)" "i(t)" "d(t)"], xlabel = "t",
legend = [:bottomright :topright :topright :topleft],
label = ["Early" "Early" "Early" "Early"])
42.5. ENDING LOCKDOWN

plot!(sol_late, vars = [5, 1, 2, 4],
legend = [:bottomright :topright :topright :topleft],
label = ["Late" "Late" "Late" "Late"])

Out[14]:

However, note that this masks highly volatile values induced by the in $R_0$ variation, as seen in the ensemble

In [15]: trajectories = 400
saveat = 1.0
ensemble_sol_early = solve(EnsembleProblem(prob_early), SOSRI(),
EnsembleThreads(), trajectories = trajectories,)

ensemble_sol_late = solve(EnsembleProblem(prob_late), SOSRI(),
EnsembleThreads(), trajectories = trajectories,)

summ_early = EnsembleSummary(ensemble_sol_early)
summ_late = EnsembleSummary(ensemble_sol_late)

plot(summ_early, idxs = (5, 1, 2, 4),
title = ["R_0" "Susceptible" "Infected" "Dead"], layout = (2, 2),
size = (900, 600),
ylabel = ["R_0(t)" "s(t)" "i(t)" "d(t)"]
xlabel = "t",
legend = [:bottomright :topright :topright :topleft],
label = ["Early" "Early" "Early" "Early"]
plot!(summ_late, idxs = (5, 1, 2, 4),
legend = [:bottomright :topright :topright :topleft],
label = ["Late" "Late" "Late" "Late"])
Finally, rather than looking at the ensemble summary, we can use data directly from the ensemble to do our own analysis.

For example, evaluating at an intermediate ($t = 350$) and final time step.

```
In [16]: N = p_early.N
t_1 = 350
t_2 = p_early.T  # i.e. the last element
bins_1 = range(0.000, 0.009, length = 30)
bins_2 = 30  # number rather than grid.

hist_1 = histogram([ensemble_sol_early.u[i](t_1)[4] for i in 1:trajectories],
                   fillalpha = 0.5, normalize = :probability, legend = :topleft, bins = bins_1,
                   label = "Early", title = "Death Proportion at t = $t_1")
histogram!(hist_1, [ensemble_sol_late.u[i](t_1)[4] for i in 1:trajectories],
          label = "Late", fillalpha = 0.5, normalize = :probability, bins = bins_1)

hist_2 = histogram([ensemble_sol_early.u[i][4, end] for i in 1:trajectories],
                   fillalpha = 0.5, normalize = :probability, bins = bins_2,
                   label = "Early", title = "Death Proportion at t = $t_2")
histogram!(hist_2, [ensemble_sol_late.u[i][4, end] for i in 1:trajectories],
          label = "Late", fillalpha = 0.5, normalize = :probability, bins = bins_2)
plot(hist_1, hist_2, size = (600,600), layout = (2, 1))
```

Out[16]:
This shows that there are significant differences after a year, but by 550 days the graphs largely coincide.

In the above code given the return from `solve` on an `EnsembleProblem`, e.g.,
```python
ensemble_sol = solve(...)```

- You can access the i'th simulation as `ensemble_sol[i]`, which then has all of the standard solution handling features
- You can evaluate at a real time period, $t$, with `ensemble_sol[i](t)`. Or access the 4th element with `ensemble_sol[i](t)[4]`
- If the $t$ was not exactly one of the `saveat` values (if specified) or the adaptive timesteps (if it was not), then it will use interpolation
- Alternatively, to access the results of the ODE as a grid exactly at the timesteps, where $j$ is timestep index, use `ensemble_sol[i][j]` or the 4th element with `ensemble_sol[i][4, j]`
- Warning: unless you have chosen a `saveat` grid, the timesteps will not be aligned between simulations. That is, `ensemble_sol[i_1].t` wouldn’t match `ensemble_sol[i_2].t`. In that case, use interpolation with `ensemble_sol[i_1](t)` etc.
42.6 Reinfection

As a final experiment, consider a model where the immunity is only temporary, and individuals become susceptible again.

In particular, assume that at rate $\nu$ immunity is lost. For illustration, we will examine the case if the average immunity lasts 12 months (i.e. $1/\nu = 360$)

The transition modifies the differential equation (1) to become

\[
\begin{align*}
    ds &= (-\gamma R_0 si + \nu r) \, dt \\
    di &= (\gamma R_0 si - \gamma i) \, dt \\
    dr &= ((1 - \delta)\gamma i - \nu r) \, dt \\
    dd &= \delta \gamma i \, dt
\end{align*}
\]

This change modifies the underlying $F$ function and adds a parameter, but otherwise the model remains the same.

We will redo the “Ending Lockdown” simulation from above, where the only difference is the new transition.

In [17]:

```plaintext
function F_reinfect(x, p, t)
    s, i, r, d, R̂, δ = x
    @unpack γ, R̂̂, η, ξ, θ, δ_bar, ν = p
    return
    [-γ*R̂*s*i + ν*r; # ds/dt
     γ*R̂̂*s*i - γ*i;  # di/dt
     (1-δ)*γ*i - ν*r # dr/dt
     δ*γ*i;            # dd/dt
     η*(R̂̂(t, p) - R̂); # dR̂̂/dt
     θ*(δ_bar - δ);    # dδ̄/dt
    ]
end
```

```plaintext
p_re_gen = @with_kw ( T = 550.0, γ = 1.0 / 18, η = 1.0 / 20, 
    R̂̂_n = 1.6, R̂̂ = (t, p) -> p*R̂̂_n, 
    δ_bar = 0.01, σ = 0.03, ξ = 0.004, θ = 0.2, N = 3.3E8, ν = 1/360)

p_re_early = p_re_gen(R̂̂ = R̂̂_lift_early, η = η_experiment, σ = σ_experiment)
p_re_late = p_re_gen(R̂̂ = R̂̂_lift_late, η = η_experiment, σ = σ_experiment)

trajectories = 400
saveat = 1.0
prob_re_early = SDEProblem(F_reinfect, G, x_0, (0, p_re_early.T), p_re_early)
prob_re_late = SDEProblem(F_reinfect, G, x_0, (0, p_re_late.T), p_re_late)
ensemble_sol_re_early = solve(EnsembleProblem(prob_re_early), SOSRI(), EnsembleThreads(), trajectories = trajectories, saveat = saveat)
ensemble_sol_re_late = solve(EnsembleProblem(prob_re_late), SOSRI(), EnsembleThreads(),)
```
42.6. REINFECTION

\[
\text{trajectories} = \text{trajectories, saveat} = \text{saveat})
\]

\[
\text{summ_re_early} = \text{EnsembleSummary(ensemble_sol_re_early)}
\]

\[
\text{summ_re_late} = \text{EnsembleSummary(ensemble_sol_re_late)}
\]

Out[17]: EnsembleSolution Solution of length 551 with uType: Float64

The ensemble simulations for the \( \nu = 0 \) and \( \nu > 0 \) can be compared to see the impact in the absence of medical innovations.

In [18]: plot(summ_late, idxs = (1, 2, 3, 4),
        title = ["Susceptible" "Infected" "Recovered" "Dead"],
        layout = (2, 2), size = (900, 600),
        ylabel = ["s(t)" "i(t)" "r(t)" "d(t)"], xlabel = "t",
        legend = :topleft,
        label = ["s(t)" "i(t)" "r(t)" "d(t)"])

plot!(summ_re_late, idxs = (1, 2, 3, 4),
      legend = :topleft,
      label = ["s(t); \nu > 0" "i(t); \nu > 0" "r(t); \nu > 0" "d(t); \nu > 0"])

Out[18]:

Finally, we can examine the same early vs. late lockdown histogram

In [19]: bins_re_1 = range(0.003, 0.010, length = 50)
bins_re_2 = range(0.0085, 0.0102, length = 50)
hist_re_1 = histogram([ensemble_sol_re_early.u[i][t_1][4] for i in 1:
                       trajectories],
                       fillalpha = 0.5, normalize = :probability,
                       legend = :topleft, bins = bins_re_1,
In this case, there are significant differences between the early and late deaths and high variance.
42.6. REINFECTION

This bleak simulation has assumed that no individuals has long-term immunity and that there will be no medical advancements on that time horizon - both of which are unlikely to be true.

Nevertheless, it suggest that the timing of lifting lockdown has a more profound impact after 18 months if we allow stochastic shocks imperfect immunity.
Part VI

Multiple Agent Models
Chapter 43

Schelling’s Segregation Model

43.1 Contents

- Overview 43.2
- The Model 43.3
- Results 43.4
- Exercises 43.5
- Solutions 43.6

43.2 Overview

In 1969, Thomas C. Schelling developed a simple but striking model of racial segregation [97]. His model studies the dynamics of racially mixed neighborhoods.

Like much of Schelling’s work, the model shows how local interactions can lead to surprising aggregate structure.

In particular, it shows that relatively mild preference for neighbors of similar race can lead in aggregate to the collapse of mixed neighborhoods, and high levels of segregation.

In recognition of this and other research, Schelling was awarded the 2005 Nobel Prize in Economic Sciences (joint with Robert Aumann).

In this lecture we (in fact you) will build and run a version of Schelling’s model.

43.3 The Model

We will cover a variation of Schelling’s model that is easy to program and captures the main idea.

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 \(0 < x, y < 1\).
43.3.1 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.

43.3.2 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.

43.4 Results

Let’s have a look at the results we got when we coded and ran this model.

As discussed above, agents are initially mixed randomly together
But after several cycles they become segregated into distinct regions
In this instance, the program terminated after 4 cycles through the set of agents, indicating that all agents had reached a state of happiness.

What is striking about the pictures is how rapidly racial integration breaks down.

This is despite the fact that people in the model don’t actually mind living mixed with the other type.

Even with these preferences, the outcome is a high degree of segregation.

### 43.5 Exercises

#### 43.5.1 Exercise 1

Implement and run this simulation for yourself.

Use 250 agents of each type.
43.6 Solutions

43.6.1 Exercise 1

Here’s one solution that does the job we want. If you feel like a further exercise you can probably speed up some of the computations and then increase the number of agents.

43.6.2 Setup

In [1]: `using InstantiateFromURL`
    # optionally add arguments to force installation: instantiate = true, precompile = true
    `github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")`

In [2]: `using Parameters, Plots, LinearAlgebra, Statistics`
    `gr(fmt = :png);`

In [3]: `Agent = @with_kw (kind, location = rand(2))`
    
    `draw_location!(a) = a.location .= rand(2)`
    
    `# distance is just 2 norm: uses our subtraction function`
    `get_distance(a, agent) = norm(a.location - agent.location)`

    `function is_happy(a)`
    `    distances = [(get_distance(a, agent), agent) for agent in agents]`
    `    sort!(distances)`
    `    neighbors = [agent for (d, agent) in distances[1:neighborhood_size]]`
    `    share = mean(isequal(a.kind), other.kind for other in neighbors)`
    `    # can also do`
    `    # share = mean(isequal(a.kind),`
    `    #     first(agents[idx]) for idx in`
    `    #     partialsortperm(get_distance.(Ref(a), agents), 1:neighborhood_size))`
    
    `    return share ≥ preference`
    `end`

    `function update!(a)`
    `    # If not happy, then randomly choose new locations until happy.`
    `    while !is_happy(a)`
    `        draw_location!(a)`
    `    end`
    `end`

    `function plot_distribution(agents)`
    `    x_vals_0, y_vals_0 = zeros(0), zeros(0)`
    `    x_vals_1, y_vals_1 = zeros(0), zeros(0)`
    `    # obtain locations of each type`
    `    for agent in agents`
    `        X, y = agent.location`
    `        if agent.kind == 0`
    `            push!(x_vals_0, x)`
        `    end`

    `end`
push!(y_vals_0, y)
else
push!(x_vals_1, x)
push!(y_vals_1, y)
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)
return plot!(legend = :none)
end

Out[3]: plot_distribution (generic function with 1 method)

In [4]:
num_of_type_0 = 250
num_of_type_1 = 250
neighborhood_size = 10 # Number of agents regarded as neighbors
preference = 0.5 # Want their kind to make at least this share of the neighborhood

# Create a list of agents
agents = vcat([Agent(kind = 0) for i in 1:num_of_type_0],
              [Agent(kind = 1) for i in 1:num_of_type_1])

plot_array = Any[]

# loop until none wishes to move
while true
    push!(plot_array, plot_distribution(agents))
    no_one_moved = true
    for agent in agents
        old_location = copy(agent.location)
        update!(agent)
        if norm(old_location - agent.location) > 0
            no_one_moved = false
        end
    end
    if no_one_moved
        break
    end
end
n = length(plot_array)
plot(plot_array...,
     layout = (n, 1),
     size = (600, 400n),
     title = reshape(["Cycle $i" for i in 1:n], 1, n))

Out[4]:
Chapter 44

A Lake Model of Employment and Unemployment

44.1 Contents

- Overview 44.2
- The Model 44.3
- Implementation 44.4
- Dynamics of an Individual Worker 44.5
- Endogenous Job Finding Rate 44.6
- Exercises 44.7
- Solutions 44.8

44.2 Overview

This lecture describes what has come to be called a lake model. The lake model is a basic tool for modeling unemployment. It allows us to analyze

- flows between unemployment and employment
- how these flows influence steady state employment and unemployment rates

It is a good model for interpreting monthly labor department reports on gross and net jobs created and jobs destroyed.

The “lakes” in the model are the pools of employed and unemployed.

The “flows” between the lakes are caused by

- firing and hiring
- entry and exit from the labor force

For the first part of this lecture, the parameters governing transitions into and out of unemployment and employment are exogenous.

Later, we’ll determine some of these transition rates endogenously using the McCall search model.

We’ll also use some nifty concepts like ergodicity, which provides a fundamental link between
CHAPTER 44. A LAKE MODEL OF EMPLOYMENT AND UNEMPLOYMENT

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.

44.2.1 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.

44.3 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 \( g = b - d \).

44.3.1 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)

44.3.2 Laws of Motion for Stock Variables

We begin by constructing laws of motion for the aggregate variables \( E_t, U_t, N_t \).

Of the mass of workers \( E_t \) who are employed at date \( t \),

- \( (1 - d)E_t \) will remain in the labor force
- of these, \( (1 - \alpha)(1 - d)E_t \) will remain employed

Of the mass of workers \( U_t \) workers who are currently unemployed,

- \( (1 - d)U_t \) will remain in the labor force
• of these, \((1 - d)\lambda U_t\) will become employed

Therefore, the number of workers who will be employed at date \(t + 1\) will be

\[
E_{t+1} = (1 - d)(1 - \alpha)E_t + (1 - d)\lambda U_t
\]

A similar analysis implies

\[
U_{t+1} = (1 - d)\alpha E_t + (1 - d)(1 - \lambda)U_t + b(E_t + U_t)
\]

The value \(b(E_t + U_t)\) is the mass of new workers entering the labor force unemployed.

The total stock of workers \(N_t = E_t + U_t\) evolves as

\[
N_{t+1} = (1 + b - d)N_t = (1 + g)N_t
\]

Letting \(X_t := \begin{pmatrix} U_t \\ E_t \end{pmatrix}\), the law of motion for \(X\) is

\[
X_{t+1} = AX_t \quad \text{where} \quad A := \begin{pmatrix} (1 - d)(1 - \lambda) + b(1 - d)\alpha + b \\ (1 - d)(1 - \alpha) \end{pmatrix}
\]

This law tells us how total employment and unemployment evolve over time.

### 44.3.3 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

\[
\begin{pmatrix} U_{t+1}/N_{t+1} \\ E_{t+1}/N_{t+1} \end{pmatrix} = \frac{1}{1 + gA} \begin{pmatrix} U_t/N_t \\ E_t/N_t \end{pmatrix}
\]

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

\[
x_{t+1} = \hat{A}x_t \quad \text{where} \quad \hat{A} := \frac{1}{1 + gA}
\]

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.

### 44.4 Implementation

Let’s code up these equations.

Here’s the code:
44.4.1 Setup

In [1]: using InstantiateFromURL
       # optionally add arguments to force installation: instantiate = true,
     - precompile = true
github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

In [2]: using LinearAlgebra, Statistics
    using Distributions, Expectations, NLsolve, Parameters, Plots
    using QuantEcon, Roots, Random

In [3]: gr(fmt = :png);

In [4]: LakeModel = @with_kw (λ = 0.283, α = 0.013, b = 0.0124, d = 0.00822)

function transition_matrices(lm)
    @unpack λ, α, b, d = lm
    g = b - d
    A = [(1 - λ) * (1 - d) + b (1 - d) * α + b]
        (1 - d) * λ (1 - d) * (1 - α)
    Â = A ./ (1 + g)
    return (A = A, Â = Â)
end

function rate_steady_state(lm)
    @unpack Â = transition_matrices(lm)
    sol = fixedpoint(x -> Â * x, fill(0.5, 2))
    converged(sol) || error("Failed to converge in $(result.iterations) iterations")
    return sol.zero
end

function simulate_stock_path(lm, X0, T)
    @unpack A = transition_matrices(lm)
    X_path = zeros(eltype(X0), 2, T)
    X = copy(X0)
    for t in 1:T
        X_path[:, t] = X
        X = A * X
    end
    return X_path
end

function simulate_rate_path(lm, x0, T)
    @unpack Â = transition_matrices(lm)
    x_path = zeros(eltype(x0), 2, T)
    x = copy(x0)
    for t in 1:T
        x_path[:, t] = x
        x = Â * x
    end
    return x_path
end

Out[4]: simulate_rate_path (generic function with 1 method)

Let’s observe these matrices for the baseline model
44.4. IMPLEMENTATION

In [5]: lm = LakeModel()
   A, Â = transition_matrices(lm)
   A

Out[5]: 2×2 Array{Float64,2}:
  0.723506  0.0252931
  0.280674  0.978887

In [6]: Â

Out[6]: 2×2 Array{Float64,2}:
  0.720495  0.0251879
  0.279505  0.974812

And a revised model

In [7]: lm = LakeModel(α = 2.0)
   A, Â = transition_matrices(lm)
   A

Out[7]: 2×2 Array{Float64,2}:
  0.723506  1.99596
  0.280674 -0.99178

In [8]: Â

Out[8]: 2×2 Array{Float64,2}:
  0.720495  1.98765
  0.279505 -0.987652

44.4.2 Aggregate Dynamics

Let’s run a simulation under the default parameters (see above) starting from \(X_0 = (12, 138)\)

In [9]: 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)
   x1 = X_path[1, :]
   x2 = X_path[2, :]
   x3 = dropdims(sum(X_path, dims = 1), dims = 1)

   plt_unemp = plot(title = "Unemployment", 1:T, x1, color = :blue, lw = 2,
   grid = true,
The aggregates $E_t$ and $U_t$ don’t converge because their sum $E_t + U_t$ grows at rate $g$.

On the other hand, the vector of employment and unemployment rates $x_t$ can be in a steady state $\bar{x}$ if there exists an $\bar{x}$ such that

- $\bar{x} = \hat{A}\bar{x}$
- the components satisfy $\bar{e} + \bar{u} = 1$

This equation tells us that a steady state level $\bar{x}$ is an eigenvector of $\hat{A}$ associated with a unit eigenvalue.

We also have $x_t \to \bar{x}$ as $t \to \infty$ provided that the remaining eigenvalue of $\hat{A}$ has modulus less than 1.

This is the case for our default parameters:

```python
In [10]: lm = LakeModel()
   : A, Â = transition_matrices(lm)
```
```
e, f = eigvals(Â)
abs(e), abs(f)
```

```
Out[10]: (0.6953067378358462, 1.0)
```

Let's look at the convergence of the unemployment and employment rate to steady state levels (dashed red line)

```
In [11]: 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)

plt_unemp = plot(title ="Unemployment rate", 1:T, x_path[1, :],color = :blue, lw = 2, alpha = 0.5, grid = true, label = "")
plot!(plt_unemp, [xbar[1]], color=:red, linetype = :hline, linestyle = :dash, lw = 2, label = "")
plt_emp = plot(title = "Employment rate", 1:T, x_path[2, :],color = :blue, lw = 2, alpha = 0.5, grid = true, label = "")
plot!(plt_emp, [xbar[2]], color=:red, linetype = :hline, linestyle = :dash, lw = 2, label = "")
plot(plt_unemp, plt_emp, layout = (2, 1), size=(700, 500))
```

```
Out[11]:
```
44.5 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.

### 44.5.1 Ergodicity

Let’s look at a typical lifetime of employment-unemployment spells.

We want to compute the average amounts of time an infinitely lived worker would spend employed and unemployed.

Let

\[ \bar{s}_{u,T} := \frac{1}{T} \sum_{t=1}^{T} \mathbb{1}\{s_t = 0\} \]

and

\[ \bar{s}_{e,T} := \frac{1}{T} \sum_{t=1}^{T} \mathbb{1}\{s_t = 1\} \]

(As usual, \( \mathbb{1}\{Q\} = 1 \) if statement \( Q \) is true and 0 otherwise)

These are the fraction of time a worker spends unemployed and employed, respectively, up until period \( T \).

If \( \alpha \in (0, 1) \) and \( \lambda \in (0, 1) \), then \( P \) is ergodic, and hence we have

\[ \lim_{T \to \infty} \bar{s}_{u,T} = \psi^*[0] \quad \text{and} \quad \lim_{T \to \infty} \bar{s}_{e,T} = \psi^*[1] \]

with probability one.

Inspection tells us that \( P \) is exactly the transpose of \( \hat{A} \) under the assumption \( b = d = 0 \).

Thus, the percentages of time that an infinitely lived worker spends employed and unemployed equal the fractions of workers employed and unemployed in the steady state distribution.

### 44.5.2 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

In [12]: using QuantEcon, Roots, Random

In [13]: lm = LakeModel(d = 0, b = 0)
   T = 5000
   # Simulation length
@unpack α, λ = lm
P = [[(1 - λ)  λ
     α    (1 - α)]]

Out[13]:
2x2 Array{Float64,2}:
0.717  0.283
0.013  0.987

In [14]:
Random.seed!(42)
mc = MarkovChain(P, [0; 1])  # 0=unemployed, 1=employed
xbar = rate_steady_state(lm)
s_path = simulate(mc, T; init=2)
ś_e = cumsum(s_path) ./ (1:T)
s_u = 1 .- s_e
s_bars = [s_u  ś_e]

plt_unemp = plot(title = "Percent of time unemployed", 1:T, s_bars[:,1],color = :blue,
lw = 2,
alpha = 0.5, label = "", grid = true)
plot!(plt_unemp, [xbar[1]], linetype = :hline, linestyle = :dash, color=:red,
lw = 2,
alpha = 0.5, label = "", grid = true)
plt_emp = plot(title = "Percent of time employed", 1:T, s_bars[:,2],color=:blue,
lw = 2,
alpha = 0.5, label = "", grid = true)
plot!(plt_emp, [xbar[2]], linetype = :hline, linestyle = :dash, color=:red,
lw = 2,
label = "")
plot(plt_unemp, plt_emp, layout = (2, 1), size=(700,500))
44.6 ENDOGENOUS JOB FINDING RATE

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.

44.6 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 [76]. All details relevant to the following discussion can be found in our treatment of that model.

44.6.1 Reservation Wage

The most important thing to remember about the model is that optimal decisions are characterized by a reservation wage $\bar{w}$.

- If the wage offer $w$ in hand is greater than or equal to $\bar{w}$, then the worker accepts.
- Otherwise, the worker rejects.

As we saw in our discussion of the model, the reservation wage depends on the wage offer distribution and the parameters

- $\alpha$, the separation rate
- $\beta$, the discount factor
- $\gamma$, the offer arrival rate
- $c$, unemployment compensation
44.6.2 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

$$\lambda = \gamma P\{w_t \geq \bar{w}\} = \gamma \sum_{w' \geq \bar{w}} p(w')$$  \hspace{1cm} (1)

44.6.3 Fiscal Policy

We can use the McCall search version of the Lake Model to find an optimal level of unemployment insurance. We assume that the government sets unemployment compensation $c$. The government imposes a lump sum tax $\tau$ sufficient to finance total unemployment payments. To attain a balanced budget at a steady state, taxes, the steady state unemployment rate $u$, and the unemployment compensation rate must satisfy

$$\tau = uc$$

The lump sum tax applies to everyone, including unemployed workers. Thus, the post-tax income of an employed worker with wage $w$ is $w - \tau$. The post-tax income of an unemployed worker is $c - \tau$.

For each specification $(c, \tau)$ of government policy, we can solve for the worker’s optimal reservation wage. This determines $\lambda$ via (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

$$W := eE[V | employed] + uU$$

where the notation $V$ and $U$ is as defined in the McCall search model lecture. The wage offer distribution will be a discretized version of the lognormal distribution $LN(\log(20), 1)$, as shown in the next figure.
We take a period to be a month.

We set \( b \) and \( d \) to match monthly birth and death rates, respectively, in the U.S. population

- \( b = 0.0124 \)
- \( d = 0.00822 \)

Following [19], we set \( \alpha \), the hazard rate of leaving employment, to

- \( \alpha = 0.013 \)

### 44.6.4 Fiscal Policy Code

We will make use of (with some tweaks) the code we wrote in the McCall model lecture, embedded below for convenience.

```plaintext
In [15]: function solve_mccall_model(mcm; U_iv = 1.0, V_iv = ones(length(mcm.w)), ᵐtol = 1e-5, iter = 2_000)
    @unpack α, β, σ, c, γ, w, E, u = mcm

    # necessary objects
    u_w = u.(w, σ)
    u_c = u(c, σ)

    # Bellman operator \( T \). Fixed point is \( x^* \) s.t. \( T(x^*) = x^* \)
    function T(x)
        V = x[1:end-1]
        U = x[end]
        [u_w + β * ((1 - α) * V .+ α * U); u_c + β * (1 - γ) * U + β * γ * E]
        end

    # value function iteration
    x_iv = [V_iv; U_iv] # initial x val
    xstar = fixedpoint(T, x_iv, iterations = iter, xtol = tol, m = 0).zero
    V = xstar[1:end-1]
```
U = xstar[end]

# compute the reservation wage
w_barindex = searchsortedfirst(V .- U, 0.0)
if w_barindex >= length(w)  # if this is true, you never want to accept
    w̄ = Inf
else
    w̄ = w[w_barindex]  # otherwise, return the number
end

# return a NamedTuple, so we can select values by name
return (V = V, U = U, w̄ = w̄)
end

Out[15]: solve_mccall_model (generic function with 1 method)

And the McCall object

In [16]: # a default utility function
u(c, σ) = c > 0 ? (c^(1 - σ) - 1) / (1 - σ) : -10e-6

# model constructor
McCallModel = @with_kw (α = 0.2,
    β = 0.98, # discount rate
    γ = 0.7,
    c = 6.0, # unemployment compensation
    σ = 2.0,
    u = u, # utility function
    w = range(10, 20, length = 60), # wage values
    E = Expectation(BetaBinomial(59, 600, 400))) #

distribution over wage values

Out[16]: ##NamedTuple_kw#267 (generic function with 2 methods)

Now let’s compute and plot welfare, employment, unemployment, and tax revenue as a function of the unemployment compensation rate

In [17]: # some global variables that will stay constant
α = 0.013
α₉₅ = (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 = range(1e-3, max_wage, length = wage_grid_size + 1)

logw_dist = Normal(log(log_wage_mean), 1)
cdf_logw = cdf.(logw_dist, log.(w_vec))
p_vec = pdf_logw ./ sum(pdf_logw)
w_vec = (w_vec[1:end-1] + w_vec[2:end]) / 2

E = expectation(Categorical(p_vec))  # expectation object

function compute_optimal_quantities(c, τ)
mcm = McCallModel(α = α_q,
    β = β,
    γ = γ,
    c = c - τ,  # post-tax compensation
    σ = σ,
    w = w_vec .- τ,  # post-tax wages
    E = E)  # expectation operator

    @unpack V, U, w̅ = solve_mccall_model(mcm)

    indicator = wage -> wage > w̅
    λ = γ * E * indicator.(w_vec .- τ)

    return w̅, λ, V, U
end

function compute_steady_state_quantities(c, τ)
w̄, λ_param, V, U = compute_optimal_quantities(c, τ)

    # compute steady state employment and unemployment rates
    lm = LakeModel(λ = λ_param, α = α_q, b = b_param, d = d_param)
x = rate_steady_state(lm)
u_rate, e_rate = x

    # compute steady state welfare
    indicator(wage) = wage > w̄
    indicator(wage) = wage > w̄
decisions = indicator.(w_vec .- τ)
w = (E * (V .* decisions)) / (E * decisions)
welfare = e_rate .* w + u_rate .* U

    return u_rate, e_rate, welfare
end

function find_balanced_budget_tax(c)

    function steady_state_budget(t)
        u_rate, e_rate, w = compute_steady_state_quantities(c, t)
        return t - u_rate * c
    end

    τ = find_zero(steady_state_budget, (0.0, 0.9c))
    return τ
end

# levels of unemployment insurance we wish to study
Nc = 60
C_vec = range(5, 140, length = Nc)
tax_vec = zeros(Nc)
unempl_vec = similar(tax_vec)
empl_vec = similar(tax_vec)
welfare_vec = similar(tax_vec)

for i in 1:Nc
    t = find_balanced_budget_tax(c_vec[i])
    u_rate, e_rate, welfare = compute_steady_state_quantities(c_vec[i], t)
    tax_vec[i] = t
    unempl_vec[i] = u_rate
    empl_vec[i] = e_rate
    welfare_vec[i] = welfare
end

plt_unemp = plot(title = "Unemployment", c_vec, unempl_vec, color = :blue, lw = 2, alpha=0.7, label = "", grid = true)
plt_tax = plot(title = "Tax", c_vec, tax_vec, color = :blue, lw = 2, alpha=0.7, label = "", grid = true)
plt_emp = plot(title = "Employment", c_vec, empl_vec, color = :blue, lw = 2, alpha=0.7, label = "", grid = true)
plt_welf = plot(title = "Welfare", c_vec, welfare_vec, color = :blue, lw = 2, alpha=0.7, label = "", grid = true)

plot(plt_unemp, plt_emp, plt_tax, plt_welf, layout = (2,2), size = (800, 700))

Out[17]:
Welfare first increases and then decreases as unemployment benefits rise.
The level that maximizes steady state welfare is approximately 62.

44.7 Exercises

44.7.1 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 [19])

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?
44.7.2 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?

44.8 Solutions

44.8.1 Exercise 1

We begin by constructing an object containing the default parameters and assigning the steady state values to $x0$

```
In [18]: lm = LakeModel()
x0 = rate_steady_state(lm)
print("Initial Steady State: $x0")

Initial Steady State: [0.08266626766923271, 0.9173337323307673]
```

Initialize the simulation values

```
In [19]: N0 = 100
   T = 50

Out[19]: 50
```

New legislation changes $\lambda$ to 0.2

```
In [20]: lm = LakeModel($\lambda = 0.2$)

Out[20]: ($\lambda = 0.2$, $\alpha = 0.013$, $b = 0.0124$, $d = 0.00822$)
```

```
In [21]: 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)
print("New Steady State: $xbar")

New Steady State: [0.11309294549489424, 0.8869070545051054]
```

Now plot stocks
In [22]:
   \[ x_1 = X_{\text{path}}[1, :) \]
   \[ x_2 = X_{\text{path}}[2, :) \]
   \[ x_3 = \text{dropdims(} \sum(X_{\text{path}}, \text{dims = 1}) \text{, dims = 1)} \]

   plt_unemp = plot(title = "Unemployment", 1:T, x1, color = :blue, grid = true,
                    label = "", bg_inside = :lightgrey)
   plt_emp = plot(title = "Employment", 1:T, x2, color = :blue, grid = true,
                   label = "", bg_inside = :lightgrey)
   plt_labor = plot(title = "Labor force", 1:T, x3, color = :blue, grid = true,
                    label = "", bg_inside = :lightgrey)

   plot(plt_unemp, plt_emp, plt_labor, layout = (3, 1), size = (800, 600))

Out[22]:

And how the rates evolve

In [23]:
   plt_unemp = plot(title = "Unemployment rate", 1:T, x_{\text{path}}[1,:], color = :blue, grid = true,
                    label = "", bg_inside = :lightgrey)
   plot!(plt_unemp, [xbar[1]], linetype = :hline, linestyle = :dash, color = :red, label = "")
We see that it takes 20 periods for the economy to converge to it’s new steady state levels.

44.8.2 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
Here are the other parameters:

\[
\begin{align*}
\text{In [25]: } & \quad b = 0.003 \\
& \quad T = 20
\end{align*}
\]

\textbf{Out[25]:} 20

Let's increase \(b\) to the new value and simulate for 20 periods

\[
\begin{align*}
\text{In [26]: } & \quad \text{lm } = \text{LakeModel}(b = b) \\
& \quad X\_path1 = \text{simulate_stock_path}(\text{lm}, x0 \ast N0, T) \quad \# \text{simulate stocks} \\
& \quad x\_path1 = \text{simulate_rate_path}(\text{lm}, x0, T) \quad \# \text{simulate rates}
\end{align*}
\]

\textbf{Out[26]:} 2×20 Array{Float64,2}:

\[
\begin{align*}
0.0826663 & \quad 0.0739981 & \quad 0.0679141 & \cdots & \quad 0.0536612 & \quad 0.0536401 & \quad 0.0536253 \\
0.917334 & \quad 0.926002 & \quad 0.932086 & & \quad 0.946339 & \quad 0.94636 & \quad 0.946375
\end{align*}
\]

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

\[
\begin{align*}
\text{In [27]: } & \quad \text{lm } = \text{LakeModel}(b = 0.0124) \\
& \quad X\_path2 = \text{simulate_stock_path}(\text{lm}, X\_path1[:, \text{end}-1], T-T+1) \quad \# \text{simulate stocks} \\
& \quad x\_path2 = \text{simulate_rate_path}(\text{lm}, x\_path1[:, \text{end}-1], T-T+1) \quad \# \text{simulate rates}
\end{align*}
\]

\textbf{Out[27]:} 2×31 Array{Float64,2}:

\[
\begin{align*}
0.0536481 & \quad 0.0624842 & \quad 0.0686335 & \cdots & \quad 0.0826652 & \quad 0.0826655 & \quad 0.0826657 \\
0.94636 & \quad 0.937516 & \quad 0.931366 & & \quad 0.917335 & \quad 0.917335 & \quad 0.917334
\end{align*}
\]

Finally we combine these two paths and plot

\[
\begin{align*}
\text{In [28]: } & \quad \text{x\_path} = \text{hcat}(\text{x\_path1, x\_path2[:, 2:end]}) \quad \# \text{note [2:] to avoid doubling period 20} \\
& \quad \text{x\_path} = \text{hcat}(\text{x\_path1, x\_path2[:, 2:end]})
\end{align*}
\]

\textbf{Out[28]:} 2×50 Array{Float64,2}:

\[
\begin{align*}
8.26663 & \quad 7.36118 & \quad 6.72069 & \quad 6.26524 & \cdots & \quad 8.45538 & \quad 8.49076 & \quad 8.52628 \\
91.7334 & \quad 92.1168 & \quad 92.238 & \quad 92.1769 & \quad 93.8293 & \quad 94.2215 & \quad 94.6153
\end{align*}
\]

\textbf{In [29]:}  

\[
\begin{align*}
x1 &= \text{X\_path}[1,:] \\
x2 &= \text{X\_path}[2,:] \\
x3 &= \text{dropdims}(\text{sum}(\text{x\_path, dims = 1}), \text{dims = 1})
\end{align*}
\]

\[
\begin{align*}
\text{plt\_unemp} &= \text{plot}(\text{title = "Unemployment"}, 1:T, x1, \text{color : blue, lw = 2, \alpha = 0.7, grid = true, label = "", bg\_inside = : lightgrey}) \\
\text{plot!}(\text{plt\_unemp, ylims = extrema(x1) .+ (-1, 1)})
\end{align*}
\]
plt_emp = plot(title = "Employment", 1:T, x2, color = :blue, lw = 2, alpha = 0.7, grid = true,
label = "", bg_inside = :lightgrey)
plot!(plt_emp, ylims = extrema(x2) .+ (-1, 1))
plt_labor = plot(title = "Labor force", 1:T, x3, color = :blue, alpha = 0.7, grid = true,
label = "", bg_inside = :lightgrey)
plot!(plt_labor, ylims = extrema(x3) .+ (-1, 1))
plot(plt_unemp, plt_emp, plt_labor, layout = (3, 1), size = (800, 600))

And the rates

plt_unemp = plot(title = "Unemployment Rate", 1:T, x_path[1,:], color = :blue, grid = true,
label = "", bg_inside = :lightgrey, lw = 2)
plot!(plt_unemp, [x0[1]], linetype = :hline, linestyle = :dash, color = :red, label = "",
w = 2)
plt_emp = plot(title = "Employment Rate", 1:T, x_path[2,:], color = :blue, grid = true,
label = "", bg_inside = :lightgrey, lw = 2)
```python
plot!(plt_emp, x[0], linetype = :hline, linestyle = :dash, color = :red, label = "", lw = 2)
plot(plt_unemp, plt_emp, layout = (2, 1), size = (800, 600))
```

Out[30]:
Chapter 45

Rational Expectations Equilibrium

45.1 Contents

• Overview 45.2
• Defining Rational Expectations Equilibrium 45.3
• Computation of an Equilibrium 45.4
• Exercises 45.5
• Solutions 45.6

“If you’re so smart, why aren’t you rich?”

45.2 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 [71].

This 1971 paper is one of a small number of research articles that kicked off the rational expectations revolution.

We follow Lucas and Prescott by employing a setting that is readily “Bellmanized” (i.e., capable of being formulated in terms of dynamic programming problems).

Because we use linear quadratic setups for demand and costs, we can adapt the LQ programming techniques described in this lecture.

We will learn about how a representative agent’s problem differs from a planner’s, and how a planning problem can be used to compute rational expectations quantities.

We will also learn about how a rational expectations equilibrium can be characterized as a fixed point of a mapping from a perceived law of motion to an actual law of motion.

Equality between a perceived and an actual law of motion for endogenous market-wide objects captures in a nutshell what the rational expectations equilibrium concept is all about.

Finally, we will learn about the important “Big K, little k” trick, a modeling device widely used in macroeconomics.

Except that for us
• Instead of “Big $K$” it will be “Big $Y$”
• Instead of “little $k$” it will be “little $y$”

45.2.1 The Big $Y$, little $y$ trick

This widely used method applies in contexts in which a “representative firm” or agent is a “price taker” operating within a competitive equilibrium.

We want to impose that

• The representative firm or individual takes aggregate $Y$ as given when it chooses individual $y$, but ....
• At the end of the day, $Y = y$, so that the representative firm is indeed representative.

The Big $Y$, little $y$ trick accomplishes these two goals by

• Taking $Y$ as beyond control when posing the choice problem of who chooses $y$; but ....
• Imposing $Y = y$ after having solved the individual’s optimization problem.

Please watch for how this strategy is applied as the lecture unfolds.

We begin by applying the Big $Y$, little $y$ trick in a very simple static context.

A simple static example of the Big $Y$, little $y$ trick

Consider a static model in which a collection of $n$ firms produce a homogeneous good that is sold in a competitive market.

Each of these $n$ firms sells output $y$.

The price $p$ of the good lies on an inverse demand curve

$$ p = a_0 - a_1 Y $$

where

• $a_i > 0$ for $i = 0,1$
• $Y = ny$ is the market-wide level of output

Each firm has total cost function

$$ c(y) = c_1 y + 0.5 c_2 y^2, \quad c_i > 0 \text{ for } i = 1, 2 $$

The profits of a representative firm are $py - c(y)$.

Using (1), we can express the problem of the representative firm as

$$ \max_y \left[ (a_0 - a_1 Y) y - c_1 y - 0.5 c_2 y^2 \right] $$

In posing problem (2), we want the firm to be a price taker.

We do that by regarding $p$ and therefore $Y$ as exogenous to the firm.

The essence of the Big $Y$, little $y$ trick is not to set $Y = ny$ before taking the first-order condition with respect to $y$ in problem (2).
This assures that the firm is a price taker.

The first order condition for problem (2) is

\[ a_0 - a_1 Y - c_1 - c_2 y = 0 \]  \tag{3}

At this point, but not before, we substitute \( Y = ny \) into (3) to obtain the following linear equation

\[ a_0 - c_1 - (a_1 + n^{-1}c_2)Y = 0 \]  \tag{4}

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 (1).

### 45.2.2 Further Reading

References for this lecture include

- [71]
- [94], chapter XIV
- [68], chapter 7

### 45.2.3 Setup

In [1]: using InstantiateFromURL
   # optionally add arguments to force installation: instantiate = true,
   # precompile = true
   github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

In [2]: using LinearAlgebra, Statistics

### 45.3 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.

### 45.3.1 Competitive Equilibrium with Adjustment Costs

To illustrate, consider a collection of \( n \) firms producing a homogeneous good that is sold in a competitive market.

Each of these \( n \) firms sells output \( y_t \).

The price \( p_t \) of the good lies on the inverse demand curve

\[
p_t = a_0 - a_1 Y_t
\]

where

- \( a_i > 0 \) for \( i = 0, 1 \)
- \( Y_t = n y_t \) is the market-wide level of output

#### The Firm’s Problem

Each firm is a price taker.

While it faces no uncertainty, it does face adjustment costs

In particular, it chooses a production plan to maximize

\[
\sum_{t=0}^{\infty} \beta^t r_t
\]

where

\[
r_t := p_t y_t - \frac{\gamma (y_{t+1} - y_t)^2}{2}, \quad y_0 \text{ given}
\]

Regarding the parameters,

- \( \beta \in (0, 1) \) is a discount factor
- \( \gamma > 0 \) measures the cost of adjusting the rate of output

Regarding timing, the firm observes \( p_t \) and \( y_t \) when it chooses \( y_{t+1} \) at at time \( t \).

To state the firm’s optimization problem completely requires that we specify dynamics for all state variables.

This includes ones that the firm cares about but does not control like \( p_t \).

We turn to this problem now.

#### Prices and Aggregate Output

In view of (5), the firm’s incentive to forecast the market price translates into an incentive to forecast aggregate output \( Y_t \).

Aggregate output depends on the choices of other firms.
We assume that \( n \) is such a large number that the output of any single firm has a negligible effect on aggregate output.

That justifies firms in regarding their forecasts of aggregate output as being unaffected by their own output decisions.

**The Firm’s Beliefs**

We suppose the firm believes that market-wide output \( Y_t \) follows the law of motion

\[
Y_{t+1} = H(Y_t)
\]

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 (8) 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

\[
v(y, Y) = \max_{y'} \left\{ a_0 y - a_1 y Y - \frac{\gamma (y' - y)^2}{2} + \beta v(y', H(Y)) \right\}
\]

Let’s denote the firm’s optimal policy function by \( h \), so that

\[
y_{t+1} = h(y_t, Y_t)
\]

where

\[
h(y, Y) := \arg\max_{y'} \left\{ a_0 y - a_1 y Y - \frac{\gamma (y' - y)^2}{2} + \beta v(y', H(Y)) \right\}
\]

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

\[
-\gamma(y' - y) + \beta v_y(y', H(Y)) = 0
\]

An important useful envelope result of Benveniste-Scheinkman [10] implies that to differentiate \( v \) with respect to \( y \) we can naively differentiate the right side of (9), giving
\[ v_y(y, Y) = a_0 - a_1 Y + \gamma (y' - y) \]

Substituting this equation into (12) gives the Euler equation

\[ -\gamma (y_{t+1} - y_t) + \beta [a_0 - a_1 Y_{t+1} + \gamma (y_{t+2} - y_{t+1})] = 0 \]  

(13)

The firm optimally sets an output path that satisfies (13), taking (8) as given, and subject to

- the initial conditions for \((y_0, Y_0)\)
- the terminal condition \(\lim_{t \to \infty} \beta^t y_t v_y(y_t, Y_t) = 0\)

This last condition is called the transversality condition, and acts as a first-order necessary condition “at infinity”.

The firm’s decision rule solves the difference equation (13) subject to the given initial condition \(y_0\) and the transversality condition.

Note that solving the Bellman equation (9) for \(v\) and then \(h\) in (11) yields a decision rule that automatically imposes both the Euler equation (13) 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

\[ Y_{t+1} = nh(Y_t/n, Y_t) \]  

(14)

Thus, when firms believe that the law of motion for market-wide output is (8), their optimizing behavior makes the actual law of motion be (14).

**45.3.2 Definition of Rational Expectations Equilibrium**

A rational expectations equilibrium or recursive competitive equilibrium of the model with adjustment costs is a decision rule \(h\) and an aggregate law of motion \(H\) such that

1. Given belief \(H\), the map \(h\) is the firm’s optimal policy function.
2. The law of motion \(H\) satisfies \(H(Y) = nh(Y/n, Y)\) for all \(Y\)

Thus, a rational expectations equilibrium equates the perceived and actual laws of motion (8) and (14).

**Fixed point characterization**

As we’ve seen, the firm’s optimum problem induces a mapping \(\Phi\) from a perceived law of motion \(H\) for market-wide output to an actual law of motion \(\Phi(H)\).

The mapping \(\Phi\) is the composition of two operations, taking a perceived law of motion into a decision rule via (9)–(11), and a decision rule into an actual law via (14).

The \(H\) component of a rational expectations equilibrium is a fixed point of \(\Phi\).
45.4 Computation of an Equilibrium

Now let’s consider the problem of computing the rational expectations equilibrium.

45.4.1 Misbehavior of Φ

Readers accustomed to dynamic programming arguments might try to address this problem by choosing some guess $H_0$ for the aggregate law of motion and then iterating with $\Phi$.

Unfortunately, the mapping $\Phi$ is not a contraction. In particular, there is no guarantee that direct iterations on $\Phi$ converge Section ??.

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, [75]).

Lucas and Prescott [71] used this method to construct a rational expectations equilibrium. The details follow.

45.4.2 A Planning Problem Approach

Our plan of attack is to match the Euler equations of the market problem with those for a single-agent choice problem.

As we’ll see, this planning problem can be solved by LQ control (linear regulator).

The optimal quantities from the planning problem are rational expectations equilibrium quantities.

The rational expectations equilibrium price can be obtained as a shadow price in the planning problem.

For convenience, in this section we set $n = 1$.

We first compute a sum of consumer and producer surplus at time $t$,

$$s(Y_t, Y_{t+1}) := \int_0^{Y_t} (a_0 - a_1 x) \, dx - \frac{\gamma(Y_{t+1} - Y_t)^2}{2}$$  \hspace{1cm} (15)

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 $Y_0$.

45.4.3 Solution of the Planning Problem

Evaluating the integral in (15) 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

\[ V(Y) = \max_{Y'} \left\{ a_0 Y - \frac{a_1}{2} Y^2 - \frac{\gamma (Y' - Y)^2}{2} + \beta V(Y') \right\} \]  \hspace{1cm} (16)

The associated first order condition is

\[ -\gamma (Y' - Y) + \beta V'(Y') = 0 \]  \hspace{1cm} (17)

Applying the same Benveniste-Scheinkman formula gives

\[ V'(Y) = a_0 - a_1 Y + \gamma (Y' - Y) \]

Substituting this into equation (17) and rearranging leads to the Euler equation

\[ \beta a_0 + \gamma Y_t - [\beta a_1 + \gamma (1 + \beta)] Y_{t+1} + \gamma \beta Y_{t+2} = 0 \]  \hspace{1cm} (18)

45.4.4 The Key Insight

Return to equation (13) and set \( y_t = Y_t \) for all \( t \).

(Recall that for this section we’ve set \( n = 1 \) to simplify the calculations)

A small amount of algebra will convince you that when \( y_t = Y_t \), equations (18) and (13) are identical.

Thus, the Euler equation for the planning problem matches the second-order difference equation that we derived by

1. finding the Euler equation of the representative firm and
2. substituting into it the expression \( Y_t = ny_t \) that “makes the representative firm be representative”

If it is appropriate to apply the same terminal conditions for these two difference equations, which it is, then we have verified that a solution of the planning problem is also a rational expectations equilibrium quantity sequence

It follows that for this example we can compute equilibrium quantities by forming the optimal linear regulator problem corresponding to the Bellman equation (16).

The optimal policy function for the planning problem is the aggregate law of motion \( H \) that the representative firm faces within a rational expectations equilibrium.

Structure of the Law of Motion

As you are asked to show in the exercises, the fact that the planner’s problem is an LQ problem implies an optimal policy — and hence aggregate law of motion — taking the form

\[ Y_{t+1} = \kappa_0 + \kappa_1 Y_t \]  \hspace{1cm} (19)
for some parameter pair $\kappa_0, \kappa_1$.

Now that we know the aggregate law of motion is linear, we can see from the firm’s Bellman equation (9) that the firm’s problem can also be framed as an LQ problem.

As you’re asked to show in the exercises, the LQ formulation of the firm’s problem implies a law of motion that looks as follows

$$y_{t+1} = h_0 + h_1 y_t + h_2 Y_t$$

Hence a rational expectations equilibrium will be defined by the parameters $(\kappa_0, \kappa_1, h_0, h_1, h_2)$ in (19)–(20).

45.5 Exercises

45.5.1 Exercise 1

Consider the firm problem described above.

Let the firm’s belief function $H$ be as given in (19).

Formulate the firm’s problem as a discounted optimal linear regulator problem, being careful to describe all of the objects needed.

Use the type LQ from the QuantEcon.jl package to solve the firm’s problem for the following parameter values:

$$a_0 = 100, a_1 = 0.05, \beta = 0.95, \gamma = 10, \kappa_0 = 95.5, \kappa_1 = 0.95$$

Express the solution of the firm’s problem in the form (20) and give the values for each $h_j$.

If there were $n$ identical competitive firms all behaving according to (20), what would (20) imply for the actual law of motion (8) for market supply.

45.5.2 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 (19)).

Extending the program that you wrote for exercise 1, determine which if any satisfy the definition of a rational expectations equilibrium

- $(94.0886298678, 0.923409232937)$
- $(93.2119845412, 0.984323478873)$
- $(95.0818452486, 0.952459076301)$

Describe an iterative algorithm that uses the program that you wrote for exercise 1 to compute a rational expectations equilibrium.

(You are not being asked actually to use the algorithm you are suggesting)

45.5.3 Exercise 3

Recall the planner’s problem described above
1. Formulate the planner’s problem as an LQ problem.
2. Solve it using the same parameter values in exercise 1
   \[ a_0 = 100, a_1 = 0.05, \beta = 0.95, \gamma = 10 \]
1. Represent the solution in the form \( Y_{t+1} = \kappa_0 + \kappa_1 Y_t \).
2. Compare your answer with the results from exercise 2.

45.5.4 Exercise 4

A monopolist faces the industry demand curve (5) 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.

45.6 Solutions

45.6.1 Exercise 1

In [3]: using QuantEcon, Printf, LinearAlgebra

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
\[
A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \kappa_1 & \kappa_0 \\ 0 & 0 & 1 \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad R = \begin{bmatrix} 0 & a_1/2 & -a_0/2 \\ a_1/2 & 0 & 0 \\ -a_0/2 & 0 & 0 \end{bmatrix}, \quad Q = \gamma/2
\]

By multiplying out you can confirm that
45.6. SOLUTIONS

- $x_t' R x_t + u_t' Q u_t = -r_t$
- $x_{t+1} = Ax_t + B u_t$

We’ll 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

$$y_{t+1} - y_t = -F_0 y_t - F_1 Y_t - F_2$$

Matching parameters with $y_{t+1} = h_0 + h_1 y_t + h_2 Y_t$ leads to

$$h_0 = -F_2, \quad h_1 = 1 - F_0, \quad h_2 = -F_1$$

Here’s our solution

In [4]: # model parameters
   a0 = 100
   a1 = 0.05
   beta = 0.95
   gamma = 10.0

   # beliefs
   x0 = 95.5
   x1 = 0.95

   # formulate the LQ problem
   A = 
   [1 0 0
    0 x1 x0
    0 0 1]

   B = [1.0, 0.0, 0.0]

   R = 
   [ 0  a1/2  -a0/2
    a1/2  0  0
    -a0/2  0  0]

   Q = 0.5 * gamma

   # solve for the optimal policy
   lq = QuantEcon.LQ(Q, R, A, B; beta = beta)
   P, F, d = stationary_values(lq)

   hh = h0, h1, h2 = -F[3], 1 - F[1], -F[2]

   @printf("F = [%.3f, %.3f, %.3f]
          F[1], F[2], F[3])")
   @printf("(h0, h1, h2) = [%.3f, %.3f, %.3f]
          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

$$Y_{t+1} = n (96.949 + y_t - 0.046 Y_t) = n 96.949 + (1 - n0.046)Y_t$$

### 45.6.2 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

```python
In [5]: candidates = ([94.0886298678, 0.92349232937],
                    [93.211845412, 0.984323478873],
                    [95.0818452486, 0.952459076301])

for (k0, k1) in candidates
    A = [1 0 0
         0 k1 k0
         0 0 1]
    lq = QuantEcon.LQ(Q, R, A, B; bet=β)
    P, F, 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)
        @printf("Equilibrium pair = (%.6f, %.6f)\n", k0, k1)
        @printf("(h0, h1, h2) = [%.6f, %.6f, %.6f]\n", h0, h1, h2)
    end
end

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)
45.6.3 Exercise 3

We are asked to write the planner problem as an LQ problem.

For the state and control vectors we choose

\[ x_t = \begin{bmatrix} Y_t \\ 1 \end{bmatrix}, \quad u_t = Y_{t+1} - Y_t \]

For the LQ matrices we set

\[ A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad B = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad R = \begin{bmatrix} a_1/2 & -a_0/2 \\ -a_0/2 & 0 \end{bmatrix}, \quad Q = \gamma/2 \]

By multiplying out you can confirm that

- \[ x_t' R x_t + u_t' Q u_t = -s(Y_t, Y_{t+1}) \]
- \[ x_{t+1} = A x_t + B u_t \]

By obtaining the optimal policy and using \( u_t = -F x_t \) or

\[ Y_{t+1} - Y_t = -F_0 Y_t - F_1 \]

we can obtain the implied aggregate law of motion via \( \kappa_0 = -F_1 \) and \( \kappa_1 = 1 - F_0 \)

```python
In [6]: # formulate the planner's LQ problem
A = I + zeros(2, 2)
B = [1.0, 0.0]
R = [ a1 / 2.0, -a0 / 2.0
     -a0 / 2.0, 0.0]
Q = \gamma / 2.0

# solve for the optimal policy
lq = QuantEcon.LQ(Q, R, A, B; bet=\beta)
P, F, d = stationary_values(lq)

# print the results
x0, x1 = -F[2], 1 - F[1]
println("x0=$x0\ \ \ \ \ \ x1=$x1")
println("x0=$x0187459215002 \ \ \ \ \ \ x1=0.9524590627039248")
```

The output yields the same \((\kappa_0, \kappa_1)\) pair obtained as an equilibrium from the previous exercise.

45.6.4 Exercise 4

The monopolist’s LQ problem is almost identical to the planner’s problem from the previous exercise, except that
\[
R = \begin{bmatrix}
\frac{a_1}{2} & -\frac{a_0}{2} \\
-\frac{a_0}{2} & 0
\end{bmatrix}
\]

The problem can be solved as follows

In [7]: # formulate the monopolist's LQ problem
\[
A = I + \text{zeros}(2, 2)
\]
\[
B = [1.0, 0.0]
\]
\[
R = \begin{bmatrix}
a_1 & \frac{-a_0}{2} \\
\frac{-a_0}{2} & 0.0
\end{bmatrix}
\]
\[
Q = \frac{\gamma}{2.0}
\]

# solve for the optimal policy
\[
lq = \text{QuantEcon.LQ}(Q, R, A, B; \text{bet}=\beta)
\]
\[
P, F, d = \text{stationary_values}(lq)
\]

# 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

Footnotes

[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 [73] and [29] 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.
Chapter 46

Markov Perfect Equilibrium

46.1 Contents

- Overview 46.2
- Background 46.3
- Linear Markov perfect equilibria 46.4
- Application 46.5
- Exercises 46.6
- Solutions 46.7

46.2 Overview

This lecture describes the concept of Markov perfect equilibrium.

Markov perfect equilibrium is a key notion for analyzing economic problems involving dy-
namic 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 [68].

46.2.1 Setup

In [1]: using InstantiateFromURL

    # optionally add arguments to force installation: instantiate = true,[]
    ~precompile = true

    github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

In [2]: using LinearAlgebra, Statistics, QuantEcon
46.3 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., [28], [93], [24]).
- Rate of extraction from a shared natural resource, such as a fishery (e.g., [67], [105]).

Let’s examine a model of the first type.

46.3.1 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

\[ p = a_0 - a_1(q_1 + q_2) \]  

(1)

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 (1) 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:

\[ \pi_i = pq_i - \gamma(\hat{q}_i - q_i)^2, \quad \gamma > 0, \]  

(2)

Substituting the inverse demand curve (1) into (2) lets us express the one-period payoff as

\[ \pi_i(q_i, q_{-i}, \hat{q}_i) = a_0q_i - a_1q_i^2 - a_1q_iq_{-i} - \gamma(\hat{q}_i - q_i)^2, \]  

(3)

where \( q_{-i} \) denotes the output of the firm other than \( i \).

The objective of the firm is to maximize \( \sum_{t=0}^{\infty} \beta^t \pi_{it} \).
Firm $i$ chooses a decision rule that sets next period quantity $\hat{q}_i$ as a function $f_i$ of the current state $(q_i, q_{-i})$.

An essential aspect of a Markov perfect equilibrium is that each firm takes the decision rule of the other firm as known and given.

Given $f_{-i}$, the Bellman equation of firm $i$ is

$$v_i(q_i, q_{-i}) = \max_{\hat{q}_i} \{ \pi_i(q_i, q_{-i}, \hat{q}_i) + \beta v_i(\hat{q}_i, f_{-i}(q_{-i}, q_i)) \} \quad (4)$$

**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).
- The maximizer on the right side of (4) 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.

**46.3.2 Computation**

One strategy for computing a Markov perfect equilibrium is iterating to convergence on pairs of Bellman equations and decision rules.

In particular, let $v_i^j, f_i^j$ be the value function and policy function for firm $i$ at the $j$-th iteration.

Imagine constructing the iterates

$$v_i^{j+1}(q_i, q_{-i}) = \max_{\hat{q}_i} \{ \pi_i(q_i, q_{-i}, \hat{q}_i) + \beta v_i^j(\hat{q}_i, f_{-i}(q_{-i}, q_i)) \} \quad (5)$$

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.

**46.4 Linear Markov perfect equilibria**

As we saw in the duopoly example, the study of Markov perfect equilibria in games with two players leads us to an interrelated pair of Bellman equations.

In linear quadratic dynamic games, these “stacked Bellman equations” become “stacked Riccati equations” with a tractable mathematical structure.

We’ll lay out that structure in a general setup and then apply it to some simple problems.
46.4.1 Coupled linear regulator problems

We consider a general linear quadratic regulator game with two players. For convenience, we’ll start with a finite horizon formulation, where $t_0$ is the initial date and $t_1$ is the common terminal date.

Player $i$ takes $\{u_{-i}\}$ as given and minimizes

$$\sum_{t=t_0}^{t_1-1} \beta^{t-t_0} \{x_t'R_t x_t + u_{1t}'Q_{11}u_{1t} + u_{2t}'S_{12}u_{2t} + 2x_t'W_{1t}u_{1t} + 2u_{2t}'M_{21}u_{2t}\}$$

while the state evolves according to

$$x_{t+1} = Ax_t + B_1u_{1t} + B_2u_{2t}$$

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 $n \times k_i$

46.4.2 Computing Equilibrium

We formulate a linear Markov perfect equilibrium as follows.

Player $i$ employs linear decision rules $u_{it} = -F_{it} x_t$, where $F_{it}$ is a $k_i \times n$ matrix.

A Markov perfect equilibrium is a pair of sequences $\{F_{1t}, F_{2t}\}$ over $t = t_0, \ldots, t_1 - 1$ such that

- $\{F_{1t}\}$ solves player 1’s problem, taking $\{F_{2t}\}$ as given, and
- $\{F_{2t}\}$ solves player 2’s problem, taking $\{F_{1t}\}$ as given

If we take $u_{2t} = -F_{2t} x_t$ and substitute it into (6) and (7), then player 1’s problem becomes minimization of

$$\sum_{t=t_0}^{t_1-1} \beta^{t-t_0} \{x_t'\Pi_{1t} x_t + u_{1t}'Q_{11}u_{1t} + 2u_{1t}'\Gamma_{11}x_t\}$$

subject to

$$x_{t+1} = A_{1t} x_t + B_1 u_{1t},$$

where

- $A_{1t} := A - B_{-1} F_{-1t}$
- $\Pi_{1t} := R_i + F_{-1t}' S_i F_{-1t}$
- $\Gamma_{11} := W_{1t}' - M_{1t}' F_{-1t}$
This is an LQ dynamic programming problem that can be solved by working backwards. The policy rule that solves this problem is
\[
F_{1t} = (Q_1 + \beta B_1' P_{1t+1} B_1)^{-1} (\beta B_1' P_{1t+1} \Lambda_{1t} + \Gamma_{1t}) \tag{10}
\]
where \(P_{1t}\) solves the matrix Riccati difference equation
\[
P_{1t} = \Pi_{1t} - (\beta B_1' P_{1t+1} \Lambda_{1t} + \Gamma_{1t})'(Q_1 + \beta B_1' P_{1t+1} B_1)^{-1} (\beta B_1' P_{1t+1} \Lambda_{1t} + \Gamma_{1t}) + \beta \Lambda_{1t}' P_{1t+1} \Lambda_{1t} \tag{11}
\]
Similarly, the policy that solves player 2’s problem is
\[
F_{2t} = (Q_2 + \beta B_2' P_{2t+1} B_2)^{-1} (\beta B_2' P_{2t+1} \Lambda_{2t} + \Gamma_{2t}) \tag{12}
\]
where \(P_{2t}\) solves
\[
P_{2t} = \Pi_{2t} - (\beta B_2' P_{2t+1} \Lambda_{2t} + \Gamma_{2t})'(Q_2 + \beta B_2' P_{2t+1} B_2)^{-1} (\beta B_2' P_{2t+1} \Lambda_{2t} + \Gamma_{2t}) + \beta \Lambda_{2t}' P_{2t+1} \Lambda_{2t} \tag{13}
\]
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 (10), (11), (12), and (13), and “work backwards” from time \(t_1 - 1\).
Since we’re working backwards, \(P_{1t+1}\) and \(P_{2t+1}\) are taken as given at each stage.
Moreover, since
- some terms on the right hand side of (10) contain \(F_{2t}\)
- some terms on the right hand side of (12) contain \(F_{1t}\)
we need to solve these \(k_1 + k_2\) equations simultaneously.

**Key insight**

A key insight is that equations (10) and (12) 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 (11) and (13).

**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.

**46.4.3 Implementation**

We use the function `nnash` from `QuantEcon.jl` that computes a Markov perfect equilibrium of the infinite horizon linear quadratic dynamic game in the manner described above.
46.5 Application

Let’s use these procedures to treat some applications, starting with the duopoly model.

46.5.1 A duopoly model

To map the duopoly model into coupled linear-quadratic dynamic programming problems, define the state and controls as

\[
\begin{bmatrix}
q_{1t} \\
q_{2t}
\end{bmatrix}
\quad \text{and} \quad
u_{it} := q_{i,t+1} - q_{it}, \quad i = 1, 2
\]

If we write

\[
x_t' R_i x_t + u_{it}' Q_i u_{it}
\]

where \( Q_1 = Q_2 = \gamma \),

\[
R_1 := \begin{bmatrix}
0 & -a_0 \gamma & 0 \\
-a_0 \gamma & a_1 \gamma & a_1 \gamma \\
0 & a_1 \gamma & 0
\end{bmatrix}
\quad \text{and} \quad
R_2 := \begin{bmatrix}
0 & 0 & -a_0 \gamma \\
0 & 0 & a_1 \gamma \\
-a_0 \gamma & a_1 \gamma & 0
\end{bmatrix}
\]

then we recover the one-period payoffs in expression (3).

The law of motion for the state \( x_t \) is \( x_{t+1} = Ax_t + B_1 u_{1t} + B_2 u_{2t} \) where

\[
A := \begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}, \quad
B_1 := \begin{bmatrix}
0 \\
1 \\
0
\end{bmatrix}, \quad
B_2 := \begin{bmatrix}
0 \\
0 \\
1
\end{bmatrix}
\]

The optimal decision rule of firm \( i \) will take the form \( u_{it} = -F_i x_t \), inducing the following closed loop system for the evolution of \( x \) in the Markov perfect equilibrium:

\[
x_{t+1} = (A - B_1 F_1 - B_1 F_2) x_t
\]  

46.5.2 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 following code

In [3]: using QuantEcon, LinearAlgebra

# parameters
46.5. APPLICATION

\[
\begin{align*}
a_0 &= 10.0 \\
a_1 &= 2.0 \\
\beta &= 0.96 \\
\gamma &= 12.0
\end{align*}
\]

# in LQ form
\[
\begin{align*}
A &= I + \text{zeros}(3, 3) \\
B_1 &= [0.0, 1.0, 0.0] \\
B_2 &= [0.0, 0.0, 1.0]
\end{align*}
\]

\[
\begin{align*}
R_1 &= \begin{bmatrix}
0.0 & -a_0 / 2.0 & 0.0 \\
-a_0 / 2.0 & a_1 & a_1 / 2.0 \\
0.0 & a_1 / 2.0 & 0.0
\end{bmatrix} \\
R_2 &= \begin{bmatrix}
0.0 & 0.0 & -a_0 / 2.0 \\
0.0 & 0.0 & a_1 / 2.0 \\
-a_0 / 2.0 & a_1 / 2.0 & a_1
\end{bmatrix}
\end{align*}
\]

\[
\begin{align*}
Q_1 &= Q_2 = \gamma \\
S_1 &= S_2 = W_1 = W_2 = M_1 = M_2 = 0.0
\end{align*}
\]

# solve using QE's nash function
\[
F_1, F_2, P_1, P_2 = \text{nash}(A, B_1, B_2, R_1, R_2, Q_1, Q_2, S_1, S_2, W_1, W_2, M_1, M_2, \text{beta}=\beta)
\]

# display policies
\[
\begin{align*}
\text{println}(&quot;Computed policies for firm 1 and firm 2:&quot;)
\text{println}(&quot;F1 = \$F1&quot;)
\text{println}(&quot;F2 = \$F2&quot;)
\end{align*}
\]

Running the code produces the following output.

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 F2 as computed above, plug it into (8) and (9) to get firm 1’s problem and solve it using LQ.

We hope that the resulting policy will agree with F1 as computed above

In [4]: \[
A_1 = A - (B_2 \ast F_2) \\
lq1 = \text{QuantEcon.LQ}(Q_1, R_1, A_1, B_1, \text{bet}=\beta) \\
P1_ih, F1_ih, d = \text{stationary_values}(lq1) \\
F1_ih
\]

Out[4]: \[
1\times3 \text{ Array}\{\text{Float64}\}: \begin{bmatrix}
-0.668466 & 0.295125 & 0.0758467
\end{bmatrix}
\]

This is close enough for rock and roll, as they say in the trade.

Indeed, isapprox agrees with our assessment
In [5]: isapprox(F1, F1_ih, atol=1e-7)

Out[5]: true

46.5.3 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 (14).

The following program

- imports \( F_1 \) and \( F_2 \) from the previous program along with all parameters
- computes the evolution of \( x_t \) using (14)
- extracts and plots industry output \( q_t = q_{1t} + q_{2t} \) and price \( p_t = a_0 - a_1 q_t \)

In [6]: using Plots
    gr(fmt=:png);

    n = 20
    x = zeros(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

    plt = plot(q, color=:blue, lw=2, alpha=0.75, label="total output")
    plot!(plt, p, color=:green, lw=2, alpha=0.75, label="price")
    plot!(plt, title="Output and prices, duopoly MPE")

Out[6]:

Note that the initial condition has been set to \( q_{10} = q_{20} = 1.0 \).

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.

### 46.6 Exercises

#### 46.6.1 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.

#### 46.6.2 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 [59].

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

- \( C_{it} = c_{i1} + c_{i2}I_{it}^2 + 0.5c_{i3}I_{it}^2 \)
- \( E_{it} = e_{i1} + e_{i2}q_{it} + 0.5e_{i3}q_{it}^2 \) 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

\[
S_{it} = Dp_{it} + b
\]

where

- \( S_t = [S_{1t} \quad S_{2t}]' \)
- \( D \) is a \( 2 \times 2 \) negative definite matrix and
- \( b \) is a vector of constants

Firm \( i \) maximizes the undiscounted sum

\[
\lim_{T \to \infty} \frac{1}{T} \sum_{t=0}^{T} (p_{it}S_{it} - E_{it} - C_{it})
\]

We can convert this to a linear quadratic problem by taking

\[
u_{it} = \begin{bmatrix} p_{it} \\ q_{it} \end{bmatrix} \quad \text{and} \quad x_t = \begin{bmatrix} I_{1t} \\ I_{2t} \\ 1 \end{bmatrix}
\]

Decision rules for price and quantity take the form \( u_{it} = -F_ix_t \).

The Markov perfect equilibrium of Judd’s model can be computed by filling in the matrices appropriately.

The exercise is to calculate these matrices and compute the following figures.

The first figure shows the dynamics of inventories for each firm when the parameters are

\[
\begin{align*}
\delta &= 0.02 \\
D &= \begin{bmatrix} -1 & 0.5 \\ 0.5 & -1 \end{bmatrix} \\
b &= [25, 25]
\end{align*}
\]
\[ c_1 = c_2 = [1, -2, 1] \]
\[ e_1 = e_2 = [10, 10, 3] \]

**Out[7]:** 3-element Array{Int64,1}:
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.
46.7 Solutions

46.7.1 Exercise 1

First let’s compute the duopoly MPE under the stated parameters

\[
\begin{align*}
\text{In } \text{[8]: # parameters} \\
a_0 &= 10.0 \\
a_1 &= 2.0 \\
\beta &= 0.96 \\
\gamma &= 12.0 \\
\end{align*}
\]

\textit{# in LQ form}

\[
\begin{align*}
A &= I + \text{zeros}(3, 3) \\
B_1 &= [0.0, 1.0, 0.0] \\
B_2 &= [0.0, 0.0, 1.0] \\
R_1 &= \begin{bmatrix}
0.0 & -a_0 / 2.0 & 0.0 \\
-a_0 / 2.0 & a_1 & a_1 / 2.0 \\
0.0 & a_1 / 2.0 & 0.0
\end{bmatrix} \\
R_2 &= \begin{bmatrix}
0.0 & 0.0 & -a_0 / 2.0 \\
0.0 & 0.0 & a_1 / 2.0 \\
-a_0 / 2.0 & a_1 / 2.0 & a_1
\end{bmatrix} \\
Q_1 = Q_2 &= \gamma \\
S_1 = S_2 = W_1 = W_2 = M_1 = M_2 = 0.0 \\
\end{align*}
\]

\textit{# solve using QE's nnash function}

\[
F_1, F_2, P_1, P_2 = \text{nnash}(A, B_1, B_2, R_1, R_2, Q_1, Q_2, S_1, S_2, W_1, W_2, M_1, M_2, \beta=\beta)
\]

\text{Out[8]:} \begin{bmatrix}
-0.6684661455442794 & 0.295124817744414 & 0.0758464663058074191 \\
0.295124817744414 & 0.0758464663058074191 & [-100.74013291681771 -13.283701011340526; 2.4358738882344166 1.9305445296892967 1.9305445296892967 5.]
\end{bmatrix}
\]

Now we evaluate the time path of industry output and prices given initial condition \(q_{10} = q_{20} = 1\)

\text{In [9]:} \begin{align*}
\text{AF} &= A - B_1 \ast F_1 - B_2 \ast F_2 \\
n &= 20 \\
x &= \text{zeros}(3, n) \\
x[\cdot, 1] &= [1 \ 1 \ 1] \\
\text{for } t \text{ in } 1:(n-1) \\
& \quad x[\cdot, t+1] = \text{AF} \ast x[\cdot, t] \\
\text{end}
\end{align*}
Next let's have a look at the monopoly solution.

For the state and control we take

\[
\begin{align*}
    x_t &= q_t - \bar{q} & \text{and} & & u_t &= q_{t+1} - q_t
\end{align*}
\]

To convert to an LQ problem we set

\[
\begin{align*}
    R &= a_1 \quad \text{and} & & Q &= \gamma
\end{align*}
\]

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 \(q_0 = 2.0\).

In [10]:

\[
\begin{align*}
    R &= a_1 \\
    Q &= \gamma \\
    A &= B = 1 \\
    \text{lq\_alt} &= \text{QuantEcon.LQ}(Q, R, A, B, \text{bet=}\beta) \\
    P, F, d &= \text{stationary\_values}(\text{lq\_alt}) \\
    \bar{q} &= a_0 / (2.0 * a_1) \\
    qm &= \text{zeros}(n) \\
    qm[1] &= 2 \\
    x0 &= qm[1] \cdot \bar{q} \\
    x &= x0 \\
    \text{for} \ i \ \text{in} \ 2:n \\
    \quad x &= A \cdot x - B \cdot F[1] \cdot x \\
    \quad qm[i] &= \text{float}(x) + \bar{q} \\
    \end{align*}
\]

\[
\begin{align*}
    pm &= a_0 .- a_1 \cdot qm
\end{align*}
\]

Let's have a look at the different time paths

In [11]:

\[
\begin{align*}
    \text{plt\_q} &= \text{plot}(qm, \text{color=}:\text{blue}, \text{lw}=2, \text{alpha}=0.75, \text{label="monopolist output"}) \\
    \text{plot!}(\text{plt\_q, q, color=}:\text{green}, \text{lw}=2, \text{alpha}=0.75, \text{label="MPE total output"}) \\
    \text{plot!}(\text{plt\_q, xlabel="time", ylabel="output", ylim=(2.4), legend=:\text{topright}}) \\
    \text{plt\_p} &= \text{plot}(pm, \text{color=}:\text{blue}, \text{lw}=2, \text{alpha}=0.75, \text{label="monopolist price"}) \\
    \text{plot!}(\text{plt\_p, p, color=}:\text{green}, \text{lw}=2, \text{alpha}=0.75, \text{label="MPE price"}) \\
    \text{plot!}(\text{plt\_p, xlabel="time", ylabel="price", legend=:\text{topright}}) \\
    \text{plot}(\text{plt\_q, plt\_p, layout=(2,1), size=(700,600))}
\end{align*}
\]
46.7.2 Exercise 2

We treat the case $\delta = 0.02$

In [12]:
$
\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
$

Out[12]:

$0.98$

Recalling that the control and state are

$$
\begin{bmatrix}
\pi_i \\
q_i
\end{bmatrix}
\quad \text{and} \quad
\begin{bmatrix}
I_{1t} \\
I_{2t} \\
1
\end{bmatrix}
$$

we set up the matrices as follows:
In [13]: # create matrices needed to compute the Nash feedback equilibrium

\[
A = \begin{bmatrix}
\delta_1 & 0 & -\delta_1 * b[1];
0 & \delta_1 & -\delta_1 * b[2];
0 & 0 & 1
\end{bmatrix}
\]

\[
B1 = \delta_1 * \begin{bmatrix}
1 & -D[1, 1];
0 & -D[2, 1];
0 & 0
\end{bmatrix}
\]

\[
B2 = \delta_1 * \begin{bmatrix}
0 & -D[1, 2];
1 & -D[2, 2];
0 & 0
\end{bmatrix}
\]

\[
R1 = -[0.5 * c1[3]] 0 0.5 * c1[2];
0 0 0
0.5 * c1[2] 0 c1[1]]
\]

\[
R2 = -[0 0 0; 0 0.5 * c2[3] 0.5 * c2[2]; 0 0.5 * c2[2] c2[1]]
\]

\[
Q1 = [-0.5 * e1[3] 0; 0 D[1, 1]]
\]

\[
Q2 = [-0.5 * e2[3] 0; 0 D[2, 2]]
\]

S1 = zeros(2, 2)
S2 = copy(S1)

W1 = [0.0 0.0; 0.0 0.0; -0.5 * e1[2] b[1] / 2.0]
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)

Out[13]: 2×2 Array{Float64,2}:
0.0 0.0
0.0 0.25

We can now compute the equilibrium using `qe.nnash`

In [14]: 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)

Firm 1's feedback rule:
Now let’s look at the dynamics of inventories, and reproduce the graph corresponding to $\delta = 0.02$.

```python
In [15]: AF = A - B1 * F1 - B2 * F2
    n = 25
    x = zeros(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, :]
    plot(I1, color=:blue, lw=2, alpha=0.75, label="inventories, firm 1")
    plot!(I2, color=:green, lw=2, alpha=0.75, label="inventories, firm 2")
    plot!(title="delta = 0.02")
```

Out[15]:

![Graph showing inventories for two firms with $\delta = 0.02$](image)

```
Chapter 47

Asset Pricing I: Finite State Models

47.1 Contents

- Overview 47.2
- Pricing Models 47.3
- Prices in the Risk Neutral Case 47.4
- Asset Prices under Risk Aversion 47.5
- Exercises 47.6
- Solutions 47.7

“A little knowledge of geometric series goes a long way” – Robert E. Lucas, Jr.

“Asset pricing is all about covariances” – Lars Peter Hansen

47.2 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

47.3 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).

### 47.3.1 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.

\[
p_t = \beta \mathbb{E}_t[d_{t+1} + p_{t+1}] \tag{1}
\]

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 \( t \).

### 47.3.2 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 [49] and Lars Peter Hansen and Scott Richard [43] showed that in quite general settings the price of an ex-dividend asset obeys

\[
p_t = \mathbb{E}_t[m_{t+1}(d_{t+1} + p_{t+1})] \tag{2}
\]

for some stochastic discount factor \( m_{t+1} \).

The fixed discount factor \( \beta \) in (1) 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.

### 47.3.3 Asset Pricing and Covariances

Recall that, from the definition of a conditional covariance \( \text{cov}_t(x_{t+1}, y_{t+1}) \), we have

\[
\mathbb{E}_t(x_{t+1}y_{t+1}) = \text{cov}_t(x_{t+1}, y_{t+1}) + \mathbb{E}_t x_{t+1} \mathbb{E}_t y_{t+1} \tag{3}
\]

If we apply this definition to the asset pricing equation (2) 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)

It is useful to regard equation (4) as a generalization of equation (1).

- In equation (1), the stochastic discount factor \( m_{t+1} = \beta \), a constant.
- In equation (1), the covariance term \( \text{cov}_t(m_{t+1}, d_{t+1} + p_{t+1}) \) is zero because \( m_{t+1} = \beta \).

Equation (4) 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.

### 47.3.4 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 (2) by \( d_t \) to get

\[ v_t = \mathbb{E}_t \left[ m_{t+1} \frac{d_{t+1}}{d_t} (1 + v_{t+1}) \right] \] (5)

Below we’ll discuss the implication of this equation.

### 47.4 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.

#### 47.4.1 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 (1) and iterating forward gives

\[
\begin{align*}
p_t &= \beta(d + p_{t+1}) \\
&= \beta(d + \beta(d + p_{t+2})) \\
&\vdots \\
&= \beta(d + \beta d + \beta^2 d + \cdots + \beta^{k-2} d + \beta^{k-1} p_{t+k})
\end{align*}
\]
Unless prices explode in the future, this sequence converges to

\[ \bar{p} := \frac{\beta d}{1 - \beta} \]

This price is the equilibrium price in the constant dividend case.

Indeed, simple algebra shows that setting \( p_t = \bar{p} \) for all \( t \) satisfies the equilibrium condition \( p_t = \beta(d + p_{t+1}) \).

### 47.4.2 Example 2: Dividends with deterministic growth paths

Consider a growing, non-random dividend process \( d_{t+1} = gd_t \) where \( 0 < g\beta < 1 \).

While prices are not usually constant when dividends grow over time, the price dividend-ratio might be.

If we guess this, substituting \( v_t = v \) into \( (5) \) as well as our other assumptions, we get \( v = \beta g(1 + v) \).

Since \( \beta g < 1 \), we have a unique positive solution:

\[ v = \frac{\beta g}{1 - \beta g} \]

The price is then

\[ p_t = \frac{\beta g}{1 - \beta g}d_t \]

If, in this example, we take \( g = 1 + \kappa \) and let \( \rho := 1/\beta - 1 \), then the price becomes

\[ p_t = \frac{1 + \kappa}{\rho - \kappa}d_t \]

This is called the Gordon formula.

### 47.4.3 Example 3: Markov growth, risk neutral pricing

Next we consider a dividend process

\[ d_{t+1} = g_{t+1}d_t \]  \hspace{1cm} (7)

The stochastic growth factor \( \{g_t\} \) is given by

\[ g_t = g(X_t), \quad t = 1,2,... \]

where

1. \( \{X_t\} \) is a finite Markov chain with state space \( S \) and transition probabilities
47.4. PRICES IN THE RISK NEUTRAL CASE

\[ P(x, y) := \mathbb{P}\{X_{t+1} = y | X_t = x\} \quad (x, y \in S) \]

1. \( g \) is a given function on \( S \) taking positive values

You can think of

- \( S \) as \( n \) possible “states of the world” and \( X_t \) as the current state
- \( g \) as a function that maps a given state \( X_t \) into a growth factor \( g_t = g(X_t) \) for the endowment
- \( \ln g_t = \ln(d_{t+1}/d_t) \) is the growth rate of dividends

(For a refresher on notation and theory for finite Markov chains see this lecture)

The next figure shows a simulation, where

- \( \{X_t\} \) evolves as a discretized AR1 process produced using Tauchen’s method
- \( g_t = \exp(X_t) \), so that \( \ln g_t = X_t \) is the growth rate

47.4.4 Setup

In [1]: using InstantiateFromURL
   # optionally add arguments to force installation: instantiate = true, precompile = true
   github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

In [2]: using LinearAlgebra, Statistics
   using Parameters, Plots, QuantEcon
   gr(fmt = :png);

In [3]: 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 \( d_0 = 1 \)

   series = [x_series g_series d_series log.(d_series)]
   labels = ["X_{t}" "g_{t}" "d_{t}" "ln(d_{t})"]
   plot(series, layout = 4, labels = labels)

Out[3]:
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 (5) to get

$$v(X_t) = \beta \mathbb{E}_t [g(X_{t+1})(1 + v(X_{t+1}))]$$

If we condition on $X_t = x$, this becomes

$$v(x) = \beta \sum_{y \in S} g(y)(1 + v(y))P(x, y)$$

or

$$v(x) = \beta \sum_{y \in S} K(x, y)(1 + v(y)) \quad \text{where} \quad K(x, y) := g(y)P(x, y)$$

(8)

Suppose that there are $n$ possible states $x_1, \ldots, x_n$.

We can then think of (8) as $n$ stacked equations, one for each state, and write it in matrix form as

$$v = \beta K(1 + v)$$

(9)
47.4. PRICES IN THE RISK NEUTRAL CASE

Here

- $v$ is understood to be the column vector $(v(x_1), \ldots, v(x_n))'$
- $K$ is the matrix $(K(x_i, x_j))_{1 \leq i, j \leq n}$
- $\mathbb{1}$ is a column vector of ones

When does (9) have a unique solution?

From the Neumann series lemma and Gelfand's 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

$$v = (I - \beta K)^{-1} \beta K \mathbb{1} \quad (10)$$

47.4.5 Code

Let’s calculate and plot the price-dividend ratio at a set of parameters.

As before, we’ll generate $\{X_t\}$ as a discretized AR1 process and set $g_t = \exp(X_t)$.

Here’s the code, including a test of the spectral radius condition

```python
In [4]: n = 25  # size of state space
   \beta = 0.9
   mc = tauchen(n, 0.96, 0.02)

   K = mc.p .* exp.(mc.state_values)'
   v = (I - \beta * K) \ (\beta * K * ones(n, 1))

   plot(mc.state_values, v,
   lw = 2,
   ylabel = "price-dividend ratio",
   xlabel = "state",
   alpha = 0.7,
   label = "v")
```

Out[4]:
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.

47.5 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 variety of bond issued by the UK government in the 19th century).
- Call options on a consol.

47.5.1 Pricing a Lucas tree

Let’s start with a version of the celebrated asset pricing model of Robert E. Lucas, Jr. [69].

As in [69], suppose that the stochastic discount factor takes the form

\[ m_{t+1} = \beta \frac{u'(c_{t+1})}{u'(c_t)} \]  

where \( u \) is a concave utility function and \( c_t \) is time \( t \) consumption of a representative consumer.

(A derivation of this expression is given in a later lecture)
Assume the existence of an endowment that follows (7).

The asset being priced is a claim on the endowment process.

Following [69], suppose further that in equilibrium, consumption is equal to the endowment, so that \( d_t = c_t \) for all \( t \).

For utility, we’ll assume the constant relative risk aversion (CRRA) specification

\[
u(c) = \frac{c^{1-\gamma}}{1-\gamma} \text{ with } \gamma > 0 \tag{12}\]

When \( \gamma = 1 \) we let \( u(c) = \ln c \).

Inserting the CRRA specification into (11) and using \( c_t = d_t \) gives

\[
m_{t+1} = \beta \left( \frac{c_{t+1}}{c_t} \right)^{-\gamma} = \beta g_t^{-\gamma} \tag{13}\]

Substituting this into (5) 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

\[
v = \beta J (\mathbb{1} + v) \]

Assuming that the spectral radius of \( J \) is strictly less than \( \beta^{-1} \), this equation has the unique solution

\[
v = (I - \beta J)^{-1} \beta J \mathbb{1} \tag{14}\]

We will define a function `tree_price` to solve for \( v \) given parameters stored in the AssetPriceModel objects.
mc = default_mc,
n = size(mc.p)[1],
g = exp

# test stability of matrix Q
function test_stability(ap, Q)
    sr = maximum(abs, eigvals(Q))
    if sr ≥ 1 / ap.β
        msg = "Spectral radius condition failed with radius = $sr"
        throw(ArgumentError(msg))
    end
end

# price/dividend ratio of the Lucas tree
function tree_price(ap; γ = ap.γ)
    # Simplify names, set up matrices
    @unpack β, mc = ap
    P, y = mc.p, mc.state_values
    y = reshape(y, 1, ap.n)
    J = P .* ap.g.(y).^(1 - γ)

    # Make sure that a unique solution exists
    test_stability(ap, J)

    # Compute v
    v = (I - β * J) \ sum(β * J, dims = 2)

    return v
end

Out[5]: tree_price (generic function with 1 method)

Here’s a plot of v as a function of the state for several values of γ, with a positively correlated Markov process and g(x) = exp(x)

In [6]: γs = [1.2, 1.4, 1.6, 1.8, 2.0]
ap = AssetPriceModel()
states = ap.mc.state_values

lines = []
labels = []

for γ in γs
    v = tree_price(ap, γ = γ)
    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")

Out[6]:
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 (13), higher growth decreases the discount factor, lowering the weight placed on future returns.

**Special cases**

In the special case \( \gamma = 1 \), we have \( J = P \).

Recalling that \( P^i \mathbb{1} = \mathbb{1} \) for all \( i \) and applying Neumann’s geometric series lemma, we are led to

\[
v = \beta(I - \beta P)^{-1} \mathbb{1} = \beta \sum_{i=0}^{\infty} \beta^i P^i \mathbb{1} = \beta \frac{1}{1 - \beta} \mathbb{1}
\]

Thus, with log preferences, the price-dividend ratio for a Lucas tree is constant.

Alternatively, if \( \gamma = 0 \), then \( J = K \) and we recover the risk neutral solution (10).

This is as expected, since \( \gamma = 0 \) implies \( u(c) = c \) (and hence agents are risk neutral).

### 47.5.2 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 (2) with $d_t = \zeta$, or

$$p_t = \mathbb{E}_t [m_{t+1}(\zeta + p_{t+1})]$$

We maintain the stochastic discount factor (13), so this becomes

$$p_t = \mathbb{E}_t \left[ \beta g_{t+1}^{-\gamma} (\zeta + p_{t+1}) \right] \quad (15)$$

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

$$p = (I - \beta M)^{-1} \beta M \zeta 1$$ \quad (16)

The above is implemented in the function `consol_price`

In [7]: function consol_price(ap, $\zeta$
   
   # Simplify names, set up matrices
   @unpack $\beta$, $\gamma$, mc, g, n = ap
   P, y = mc.p, mc.state_values
   y = reshape(y, 1, n)
   M = P .* g.(y).^(-$\gamma$)

   # Make sure that a unique solution exists
   test_stability(ap, M)

   # Compute price
   return (I - $\beta$ * M) \ sum($\beta$ * $\zeta$ * M, dims = 2)

Out[7]: consol_price (generic function with 1 method)

47.5.3 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

$$w(X_t, p_S) = \max \left\{ \beta E_t \frac{u'(c_{t+1})}{u'(c_t)} w(X_{t+1}, p_S), p(X_t) - p_S \right\}$$

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

$$w(x, p_S) = \max \left\{ \beta \sum_{y \in S} P(x, y) g(y)^{-\gamma} w(y, p_S), p(x) - p_S \right\} \quad (17)$$

With $M(x, y) = P(x, y) g(y)^{-\gamma}$ and $w$ as the vector of values $(w(x_i), p_S)^n_{i=1}$, we can express (17) as the nonlinear vector equation

$$w = \max\{\beta M w, p - p_S \mathbf{1}\} \quad (18)$$

To solve (18), form the operator $T$ mapping vector $w$ into vector $T w$ via

$$T w = \max\{\beta M w, p - p_S \mathbf{1}\}$$

Start at some initial $w$ and iterate to convergence with $T$.

We can find the solution with the following function `call_option`

```matlab
# price of perpetual call on consol bond
function call_option(ap, ζ, p_s, ϵ = 1e-7)
    # Simplify names, set up matrices
    @unpack β, γ, mc, g, n = ap
    P, y = mc.p, mc.state_values
    y = reshape(y, 1, n)
    M = P .* 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 = ϵ + 1
while (error > ϵ)
    # 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

Out[8]: call_option (generic function with 2 methods)

Here’s a plot of $w$ compared to the consol price when $P_S = 40$

In [9]: 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")

Out[9]:
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

### 47.5.4 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 $m_{t+1} = \beta g_{t+\gamma}$.

It follows that the reciprocal $R_t^{-1}$ of the gross risk-free interest rate $R_t$ in state $x$ is

$$E_t m_{t+1} = \beta \sum_{y \in S} P(x, y) g(y)^{-\gamma}$$

We can write this as

$$m_1 = \beta M 1$$

where the $i$-th element of $m_1$ is the reciprocal of the one-period gross risk-free interest rate in state $x_i$.

**Other terms**

Let $m_j$ be an $n \times 1$ vector whose $i$th component is the reciprocal of the $j$-period gross risk-free interest rate in state $x_i$.

Then $m_1 = \beta M$, and $m_{j+1} = M m_j$ for $j \geq 1$.

### 47.6 Exercises

#### 47.6.1 Exercise 1

In the lecture, we considered **ex-dividend assets**.

A **cum-dividend** asset is a claim to the stream $d_t, d_{t+1}, ....$

Following (1), find the risk-neutral asset pricing equation for one unit of a cum-dividend asset.

With a constant, non-random dividend stream $d_t = d > 0$, what is the equilibrium price of a cum-dividend asset?

With a growing, non-random dividend process $d_t = gd_t$ where $0 < g\beta < 1$, what is the equilibrium price of a cum-dividend asset?
47.6.2 Exercise 2

Consider the following primitives

In [10]:
   
   \[
   n = 5 \\
   P = \text{fill}(0.0125, n, n) + (0.95 - 0.0125) \text{I} \\
   s = [1.05, 1.025, 1.0, 0.975, 0.95] \\
   \gamma = 2.0 \\
   \beta = 0.94 \\
   \zeta = 1.0
   \]

Out[10]: 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 \( p_S = 150.0 \)

47.6.3 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 (17).

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

\[
   w_k = \max\{\beta M w_{k-1}, p - p_S I\} \quad k = 1, 2, \ldots \quad \text{with } w_0 = 0
\]

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?
47.7 Solutions

47.7.1 Exercise 2

In [11]: n = 5
    P = fill(0.0125, n, n) + (0.95 - 0.0125)I
    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

Out[11]: 150.0

Next we'll create an instance of AssetPriceModel to feed into the functions.

In [12]: ap = AssetPriceModel(β = β, mc = mc, γ = γ, g = x -> x)

Out[12]: (β = 0.94, γ = 2.0, mc = Discrete Markov Chain
stochastic matrix of type Array{Float64,2}:
0.95 0.0125 ... 0.0125 0.0125; 0.0125 0.95 ... 0.0125 0.0125; ... ; 0.0125 0.
0.0125 ... 0.95
0.0125; 0.0125 0.0125 ... 0.0125 0.95], n = 5, g = var"#5#6"())

In [13]: v = tree_price(ap)
println("Lucas Tree Prices: $v\n")

Lucas Tree Prices: [29.474015777145; 21.935706611219704; 17.571422357262904; 14.725150017725884; 12.722217630644252]

In [14]: v_consol = consol_price(ap, 1.0)
println("Consol Bond Prices: $(v_consol)\n")

Consol Bond Prices: [753.8710047641987; 242.55144081989457; 148.67554548466478; 109.25108965624712; 87.56860138531121]

In [15]: w = call_option(ap, ζ, p_s)

Out[15]: 5x1 Array{Float64,2}:
603.8710047641987
176.839334301913
108.6773449337003
80.05179254233045
64.30843748377002
47.7.2 Exercise 3

Here's a suitable function:

```python
In [16]: function finite_horizon_call_option(ap, ζ, p_s, k)

    # Simplify names, set up matrices
    @unpack β, γ, mc = ap
    P, y = mc.p, 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

Out[16]: finite_horizon_call_option (generic function with 1 method)
```

```python
In [17]: 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)))
```

Out[17]:
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.
Chapter 48

Asset Pricing II: The Lucas Asset Pricing Model

48.1 Contents

- Overview 48.2
- The Lucas Model 48.3
- Exercises 48.4
- Solutions 48.5

48.2 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 [69] 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.

48.3 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

877
CHAPTER 48. ASSET PRICING II: THE LUCAS ASSET PRICING MODEL

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.

48.3.1 Basic Setup

Let’s review the set up.

Assets

There is a single “productive unit” that costlessly generates a sequence of consumption goods \( \{y_t\}_{t=0}^{\infty} \).
Another way to view \( \{y_t\}_{t=0}^{\infty} \) is as a consumption endowment for this economy.
We will assume that this endowment is Markovian, following the exogenous process

\[
y_{t+1} = G(y_t, \xi_{t+1})
\]

Here \( \{\xi_t\} \) is an iid shock sequence with known distribution \( \phi \) and \( y_t \geq 0 \).
An asset is a claim on all or part of this endowment stream.
The consumption goods \( \{y_t\}_{t=0}^{\infty} \) are nonstorable, so holding assets is the only way to transfer wealth into the future.
For the purposes of intuition, it’s common to think of the productive unit as a “tree” that produces fruit.
Based on this idea, a “Lucas tree” is a claim on the consumption endowment.

Consumers

A representative consumer ranks consumption streams \( \{c_t\} \) according to the time separable utility functional

\[
E \sum_{t=0}^{\infty} \beta^t u(c_t)
\]

Here
- \( \beta \in (0, 1) \) is a fixed discount factor
- \( u \) is a strictly increasing, strictly concave, continuously differentiable period utility function
- \( E \) is a mathematical expectation

48.3.2 Pricing a Lucas Tree

What is an appropriate price for a claim on the consumption endowment?
We’ll price an \textit{ex dividend} claim, meaning that
48.3. THE LUCAS MODEL

- the seller retains this period’s dividend
- the buyer pays \( p_t \) today to purchase a claim on
  \(- y_{t+1} \) and
  \(- the right to sell the claim tomorrow at price \( p_{t+1} \)

Since this is a competitive model, the first step is to pin down consumer behavior, taking prices as given.

Next we’ll impose equilibrium constraints and try to back out prices.

In the consumer problem, the consumer’s control variable is the share \( \pi_t \) of the claim held in each period.

Thus, the consumer problem is to maximize (1) 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 .
- 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 (2) and hence eliminate the constraint, obtaining
\[ v(\pi, y) = \max_{\pi'} \left\{ u[\pi(y + p(y)) - \pi'p(y)] + \beta \int v(\pi', G(y, z))\phi(dz) \right\} \]  

(3)

The solution to this dynamic programming problem is an optimal policy expressing either \( \pi' \) or \( c \) as a function of the state \((\pi, y)\).

- Each one determines the other, since \( c(\pi, y) = \pi(y + p(y)) - \pi'(\pi, y)p(y) \).

**Next steps**

What we need to do now is determine equilibrium prices.

It seems that to obtain these, we will have to

1. Solve this two dimensional dynamic programming problem for the optimal policy.
2. Impose equilibrium constraints.
3. Solve out for the price function \( p(y) \) directly.

However, as Lucas showed, there is a related but more straightforward way to do this.

**Equilibrium constraints**

Since the consumption good is not storable, in equilibrium we must have \( c_t = y_t \) for all \( t \).

In addition, since there is one representative consumer (alternatively, since all consumers are identical), there should be no trade in equilibrium.

In particular, the representative consumer owns the whole tree in every period, so \( \pi_t = 1 \) for all \( t \).

Prices must adjust to satisfy these two constraints.

**The equilibrium price function**

Now observe that the first order condition for (3) can be written as

\[ u'(c)p(y) = \beta \int v'_1(\pi', G(y, z))\phi(dz) \]

where \( v'_1 \) is the derivative of \( v \) with respect to its first argument.

To obtain \( v'_1 \) we can simply differentiate the right hand side of (3) 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

\[ p(y) = \beta \int \frac{u'[G(y, z)]}{u'(y)}[G(y, z) + p(G(y, z))]\phi(dz) \]

(4)
48.3. THE LUCAS MODEL

In sequential rather than functional notation, we can also write this as

\[ p_t = \mathbb{E}_t \left[ \beta \frac{u'(c_{t+1})}{u'(c_t)} (y_{t+1} + p_{t+1}) \right] \]  

(5)

This is the famous consumption-based asset pricing equation.

Before discussing it further we want to solve out for prices.

48.3.3 Solving the Model

Equation (4) is a functional equation in the unknown function \( p \).

The solution is an equilibrium price function \( p^* \).

Let’s look at how to obtain it.

Setting up the problem

Instead of solving for it directly we’ll follow Lucas’ indirect approach, first setting

\[ f(y) := u'(y)p(y) \]

(6)

so that (4) becomes

\[ f(y) = h(y) + \beta \int f[G(y, z)]\phi(dz) \]

(7)

Here \( h(y) := \beta \int u'[G(y, z)]G(y, z)\phi(dz) \) is a function that depends only on the primitives.

Equation (7) is a functional equation in \( f \).

The plan is to solve out for \( f \) and convert back to \( p \) via (6).

To solve (7) we’ll use a standard method: convert it to a fixed point problem.

First we introduce the operator \( T \) mapping \( f \) into \( Tf \) as defined by

\[ (Tf)(y) = h(y) + \beta \int f[G(y, z)]\phi(dz) \]

(8)

The reason we do this is that a solution to (7) 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 \( \mathcal{cb}\mathbb{R}_+ \) be the set of continuous bounded functions \( f : \mathbb{R}_+ \to \mathbb{R}_+ \).

We now show that
1. $T$ has exactly one fixed point $f^*$ in $cb\mathbb{R}_+$.

2. For any $f \in cb\mathbb{R}_+$, the sequence $T^k f$ converges uniformly to $f^*$.

(Note: If you find the mathematics heavy going you can take 1–2 as given and skip to the next section)

Recall the Banach contraction mapping theorem.

It tells us that the previous statements will be true if we can find an $\alpha < 1$ such that

$$\|Tf - Tg\| \leq \alpha \|f - g\|, \quad \forall f, g \in cb\mathbb{R}_+ \quad (9)$$

Here $\|h\| := \sup_{x \in \mathbb{R}_+} |h(x)|$.

To see that (9) is valid, pick any $f, g \in cb\mathbb{R}_+$ and any $y \in \mathbb{R}_+$.

Observe that, since integrals get larger when absolute values are moved to the inside,

$$|Tf(y) - Tg(y)| = \left| \beta \int f(G(y, z))\phi(dz) - \beta \int g(G(y, z))\phi(dz) \right|$$
$$\leq \beta \int |f(G(y, z)) - g(G(y, z))| \phi(dz)$$
$$\leq \beta \int \|f - g\| \phi(dz)$$
$$= \beta \|f - g\|$$

Since the right hand side is an upper bound, taking the sup over all $y$ on the left hand side gives (9) with $\alpha := \beta$.

### 48.3.4 Computation – An Example

The preceding discussion tells that we can compute $f^*$ by picking any arbitrary $f \in cb\mathbb{R}_+$ and then iterating with $T$.

The equilibrium price function $p^*$ can then be recovered by $p^*(y) = f^*(y)/u'(y)$.

Let’s try this when $\ln y_{t+1} = \alpha \ln y_t + \sigma \epsilon_{t+1}$ where $\{\epsilon_t\}$ is iid and standard normal.

Utility will take the isoelastic form $u(c) = c^{1-\gamma}/(1 - \gamma)$, where $\gamma > 0$ is the coefficient of relative risk aversion.

Some code to implement the iterative computational procedure can be found below:

### 48.3.5 Setup

In [1]: using InstantiateFromURL
# optionally add arguments to force installation: instantiate = true, precompile = true
github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

In [2]: using LinearAlgebra, Statistics
using Distributions, Interpolations, Parameters, Plots, QuantEcon, Random
gr(fmt = :png);
In [3]: # model
    function LucasTree(;γ = 2.0,
                  β = 0.95,
                  α = 0.9,
                  σ = 0.1,
                  grid_size = 100)

    ϕ = LogNormal(0.0, σ)
    shocks = rand(ϕ, 500)

    # build a grid with mass around stationary distribution
    ssd = σ / sqrt(1 - α^2)
    grid_min, grid_max = exp(-4ssd), exp(4ssd)
    grid = range(grid_min, grid_max, length = grid_size)

    # set h(y) = β * int u'(G(y,z)) G(y,z) φ(z) dz
    h = similar(grid)
    for (i, y) in enumerate(grid)
        h[i] = β * mean((y^α .* shocks).^(1 - γ))
    end
    return (γ = γ, β = β, α = α, σ = σ, ϕ = ϕ, grid = grid, shocks = shocks, h = h)
end

    # approximate Lucas operator, which returns the updated function Tf on the grid
    function lucas_operator(lt, f)

        # unpack input
        @unpack grid, α, β, h = lt
        z = lt.shocks

        Af = LinearInterpolation(grid, f, extrapolation_bc=Line())

        Tf = [ h[i] + β * mean(Af.(grid[i]^α .* z)) for i in 1:length(grid) ]
        return Tf
    end

    # get equilibrium price for Lucas tree
    function solve_lucas_model(lt; tol = 1e-6,
                               max_iter = 500)

        @unpack grid, γ = lt

        i = 0
        f = zero(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.^\gamma

return price
end

Out[3]: solve_lucas_model (generic function with 1 method)

An example of usage is given in the docstring and repeated here

In [4]: Random.seed!(42) # For reproducible results.

    tree = LucasTree(\gamma = 2.0, \beta = 0.95, \alpha = 0.90, \sigma = 0.1)

    price_vals = solve_lucas_model(tree);

Here’s the resulting price function

In [5]: plot(tree.grid, price_vals, lw = 2, label = "p^*(y)")

    plot!(xlabel = "y", ylabel = "price", legend = :topleft)

Out[5]:

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.

48.4 Exercises

48.4.1 Exercise 1

Replicate the figure to show how discount rates affect prices.

48.5 Solutions

In [6]: plot()
   for $\beta$ in (.95, 0.98)
       tree = LucasTree($\beta = \beta$)
       grid = tree.grid
       price_vals = solve_lucas_model(tree)
       plot!(grid, price_vals, lw = 2, label = "beta = beta_var")
   end

   plot!(xlabel = "y", ylabel = "price", legend = :topleft)
Chapter 49

Asset Pricing III: Incomplete Markets

49.1 Contents

- Overview 49.2
- Structure of the Model 49.3
- Solving the Model 49.4
- Exercises 49.5
- Solutions 49.6

49.2 Overview

This lecture describes a version of a model of Harrison and Kreps [48].

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

49.2.1 References

Prior to reading the following you might like to review our lectures on

- Markov chains
- Asset pricing with finite state space

49.2.2 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.

### 49.2.3 Setup

In [1]: using InstantiateFromURL
   # optionally add arguments to force installation: instantiate = true,
   precompile = true
github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

In [2]: using LinearAlgebra, Statistics

### 49.3 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

$$P_a = \begin{bmatrix} 1/2 & 1/2 \\ 1/4 & 3/4 \end{bmatrix}$$

Investors of type $b$ think the transition matrix is

$$P_b = \begin{bmatrix} 2/3 & 1/3 \\ 1/3 & 2/3 \end{bmatrix}$$
49.3. **STRUCTURE OF THE MODEL**

The stationary (i.e., invariant) distributions of these two matrices can be calculated as follows:

In [3]: using QuantEcon

\[
\begin{align*}
qa &= \begin{bmatrix} 1/2 & 1/2; 2/3 & 1/3 \end{bmatrix} \\
qb &= \begin{bmatrix} 2/3 & 1/3; 1/4 & 3/4 \end{bmatrix} \\
mcA &= \text{MarkovChain}(qa) \\
mcB &= \text{MarkovChain}(qb) \\
stA &= \text{stationary_distributions}(mcA)
\end{align*}
\]

Out[3]: 1-element Array{Array{Float64,1},1}:

\[
[0.5714285714285715, 0.4285714285714286]
\]

In [4]: stB = \text{stationary_distributions}(mcB)

Out[4]: 1-element Array{Array{Float64,1},1}:

\[
[0.42857142857142855, 0.5714285714285714]
\]

The stationary distribution of $P_a$ is approximately $\pi_A = [0.57, 0.43]$.

The stationary distribution of $P_b$ is approximately $\pi_B = [0.43, 0.57]$.

### 49.3.1 Ownership Rights

An owner of the asset at the end of time $t$ is entitled to the dividend at time $t + 1$ and also has the right to sell the asset at time $t + 1$.

Both types of investors are risk-neutral and both have the same fixed discount factor $\beta \in (0, 1)$.

In our numerical example, we’ll set $\beta = .75$, just as Harrison and Kreps did.

We’ll eventually study the consequences of two different assumptions about the number of shares $A$ relative to the resources that our two types of investors can invest in the stock.

1. Both types of investors have enough resources (either wealth or the capacity to borrow) so that they can purchase the entire available stock of the asset Section ??.

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.

### 49.3.2 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.
49.3.3 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 period’s dividend than a type $b$ agent.
- In state 1, a type $b$ agent is more optimistic about next period’s 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

$$P_o = \begin{bmatrix}
\frac{1}{4} & \frac{1}{4} \\
\frac{1}{4} & \frac{3}{4}
\end{bmatrix}$$

Temporarily pessimistic believe the transition matrix

$$P_p = \begin{bmatrix}
\frac{1}{4} & \frac{1}{4} \\
\frac{3}{4} & \frac{1}{4}
\end{bmatrix}$$

We’ll return to these matrices and their significance in the exercise.

49.3.4 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 $s_t$.

49.4 Solving the Model

Now let’s turn to solving the model.

This amounts to determining equilibrium prices under the different possible specifications of beliefs and constraints listed above.

In particular, we compare equilibrium price functions under the following alternative assumptions about beliefs:

1. There is only one type of agent, either $a$ or $b$.

2. There are two types of agent differentiated only by their beliefs. Each type of agent has sufficient resources to purchase all of the asset (Harrison and Kreps’s setting).
49.4. SOLVING THE MODEL

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.

49.4.1 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 Kreps’s specifications of \( P_a, P_b, \beta \).

<table>
<thead>
<tr>
<th>( s_t )</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p_a )</td>
<td>1.33</td>
<td>1.22</td>
</tr>
<tr>
<td>( p_b )</td>
<td>1.45</td>
<td>1.91</td>
</tr>
<tr>
<td>( p_o )</td>
<td>1.85</td>
<td>2.08</td>
</tr>
<tr>
<td>( p_p )</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( \hat{p}_a )</td>
<td>1.85</td>
<td>1.69</td>
</tr>
<tr>
<td>( \hat{p}_b )</td>
<td>1.69</td>
<td>2.08</td>
</tr>
</tbody>
</table>

Here

- \( p_a \) is the equilibrium price function under homogeneous beliefs \( P_a \)
- \( p_b \) is the equilibrium price function under homogeneous beliefs \( P_b \)
- \( p_o \) is the equilibrium price function under heterogeneous beliefs with optimistic marginal investors
- \( p_p \) is the equilibrium price function under heterogeneous beliefs with pessimistic marginal investors
- \( \hat{p}_a \) is the amount type \( a \) investors are willing to pay for the asset
- \( \hat{p}_b \) is the amount type \( b \) investors are willing to pay for the asset

We’ll explain these values and how they are calculated one row at a time.

49.4.2 Single Belief Prices

We’ll start by pricing the asset under homogeneous beliefs.

(This is the case treated in the lecture on asset pricing with finite Markov states)

Suppose that there is only one type of investor, either of type \( a \) or \( b \), and that this investor always “prices the asset”.

Let \( p_h = \begin{bmatrix} p_h(0) \\ p_h(1) \end{bmatrix} \) 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

\[
\begin{bmatrix} p_h(0) \\ p_h(1) \end{bmatrix} = \beta[I - \beta P_h]^{-1} P_h \begin{bmatrix} 0 \\ 1 \end{bmatrix}
\]
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

```python
In [5]: using LinearAlgebra

def price_single_beliefs(transition, dividend_payoff; 
                        β=.75)
    # First compute inverse piece
    imbq_inv = inv(I - β * transition)

    # Next compute prices
    prices = β * (imbq_inv * transition) * dividend_payoff
    return prices
end
```

Out[5]: `price_single_beliefs (generic function with 1 method)`

**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.

### 49.4.3 Pricing under Heterogeneous Beliefs

There are several cases to consider.

The first is when both types of agent have sufficient wealth to purchase all of the asset themselves.

In this case the marginal investor who prices the asset is the more optimistic type, so that the equilibrium price $\bar{p}$ satisfies Harrison and Kreps’s key equation:

$$\bar{p}(s) = \beta \max \{ P_a(s,0)\bar{p}(0) + P_a(s,1)(1 + \bar{p}(1)), \ P_b(s,0)\bar{p}(0) + P_b(s,1)(1 + \bar{p}(1)) \} \quad (2)$$

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 (2) is a functional equation that, like a Bellman equation, can be solved by

- starting with a guess for the price vector \( \bar{p} \) and
- iterating to convergence on the operator that maps a guess \( \bar{p}^j \) into an updated guess \( \bar{p}^{j+1} \) defined by the right side of (2), namely

\[
\bar{p}^{j+1}(s) = \beta \max \{ P_a(s, 0)\bar{p}^j(0) + P_a(s, 1)(1 + \bar{p}^j(1)), P_b(s, 0)\bar{p}^j(0) + P_b(s, 1)(1 + \bar{p}^j(1)) \} \tag{3}
\]

for \( s = 0, 1 \).

The third row of the table reports equilibrium prices that solve the functional equation when \( \beta = .75 \).

Here the type that is optimistic about \( s_{t+1} \) prices the asset in state \( s_t \).

It is instructive to compare these prices with the equilibrium prices for the homogeneous belief economies that solve under beliefs \( P_a \) and \( P_b \).

Equilibrium prices \( \bar{p} \) in the heterogeneous beliefs economy exceed what any prospective investor regards as the fundamental value of the asset in each possible state.

Nevertheless, the economy recurrently visits a state that makes each investor want to purchase the asset for more than he believes its future dividends are worth.

The reason is that he expects to have the option to sell the asset later to another investor who will value the asset more highly than he will.

- Investors of type \( a \) are willing to pay the following price for the asset

\[
\hat{p}_a(s) = \begin{cases} 
\bar{p}(0) & \text{if } s_t = 0 \\
\beta(P_a(1, 0)\bar{p}(0) + P_a(1, 1)(1 + \bar{p}(1))) & \text{if } s_t = 1
\end{cases}
\]

- Investors of type \( b \) are willing to pay the following price for the asset

\[
\hat{p}_b(s) = \begin{cases} 
\beta(P_b(0, 0)\bar{p}(0) + P_b(0, 1)(1 + \bar{p}(1))) & \text{if } s_t = 0 \\
\bar{p}(1) & \text{if } s_t = 1
\end{cases}
\]

Evidently, \( \hat{p}_a(1) < \bar{p}(1) \) and \( \hat{p}_b(0) < \bar{p}(0) \).

Investors of type \( a \) want to sell the asset in state 1 while investors of type \( b \) want to sell it in state 0.

- The asset changes hands whenever the state changes from 0 to 1 or from 1 to 0.
- The valuations \( \hat{p}_a(s) \) and \( \hat{p}_b(s) \) are displayed in the fourth and fifth rows of the table.
- Even the pessimistic investors who 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

In [6]: function price_optimistic_beliefs(transitions, dividend_payoff;
\text{beta}=.75, \text{max_iter}=50000, \\
\text{tol}=1e-16)
# We will guess an initial price vector of [0, 0]

```python
p_new = [0,0]
p_old = [10.0,10.0]
```

# We know this is a contraction mapping, so we can iterate to conv

```python
for i ∈ 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
```

```python
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]]
return p_new, phat_a, phat_b
```

```
Out[6]: price_optimistic_beliefs (generic function with 1 method)
```

## 49.4.4 Insufficient Funds

Outcomes differ when the more optimistic type of investor has insufficient wealth — or insufficient ability to borrow enough — to hold the entire stock of the asset.

In this case, the asset price must adjust to attract pessimistic investors.

Instead of equation (2), the equilibrium price satisfies

\[
\bar{p}(s) = \beta \min \{P_a(s,1)\bar{p}(0) + P_a(s,1)(1 + \bar{p}(1)), P_b(s,1)\bar{p}(0) + P_b(s,1)(1 + \bar{p}(1))\} \tag{4}
\]

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 \( \bar{p} \) is lower than the homogeneous belief prices \( p_a \) and \( p_b \) in both states.

When pessimistic investors price the asset according to (4), optimistic investors think that the asset is underpriced.

If they could, optimistic investors would willingly borrow at the one-period gross interest rate \( \beta^{-1} \) to purchase more of the asset.

Implicit constraints on leverage prohibit them from doing so.

When optimistic investors price the asset as in equation (2), pessimistic investors think that the asset is overpriced and would like to sell the asset short.
Constraints on short sales prevent that.

Here’s code to solve for \( \tilde{p} \) using iteration:

In [7]: function price_pessimistic_beliefs(transitions, dividend_payoff; β=.75, max_iter=50000, tol=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 = 1:max_iter
    p_old = p_new
    temp=[minimum((q * p_old) + (q^* dividend_payoff)) for q in transitions]
    p_new = β * temp

    # If we succed in converging, break out of for loop
    if maximum(sqrt, ((p_new - p_old).^2)) < 1e-12
        break
    end

end

return p_new
end

Out[7]: price_pessimistic_beliefs (generic function with 1 method)

49.4.5 Further Interpretation

[96] interprets the Harrison-Kreps model as a model of a bubble — a situation in which an asset price exceeds what every investor thinks is merited by the asset’s underlying dividend stream.

Scheinkman stresses these features of the Harrison-Kreps model:

- Compared to the homogeneous beliefs setting leading to the pricing formula, high volume occurs when the Harrison-Kreps pricing formula prevails.

Type \( a \) investors sell the entire stock of the asset to type \( b \) investors every time the state switches from \( s_t = 0 \) 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.

49.5 Exercises

49.5.1 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>$\hat{p}_a$</td>
<td>1.85</td>
<td>1.69</td>
</tr>
<tr>
<td>$\hat{p}_b$</td>
<td>1.69</td>
<td>2.08</td>
</tr>
</tbody>
</table>

You will first need to define the transition matrices and dividend payoff vector.

49.6 Solutions

49.6.1 Exercise 1

First we will obtain equilibrium price vectors with homogeneous beliefs, including when all investors are optimistic or pessimistic.

```plaintext
In [8]: 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), digits=2)
    println("State 0: \$s0")
    println("State 1: \$s1")
    println(repeat("-", 20))
end

p_a

====================
State 0: 1.33
We will use the `price_optimistic_beliefs` function to find the price under heterogeneous beliefs.

```python
In [9]: 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, digits = 2)
    println("State 0: \$s0")
    println("State 1: \$s1")
    println(repeat("-", 20))
```

Notice that the equilibrium price with heterogeneous beliefs is equal to the price under single beliefs with optimistic investors - this is due to the marginal investor being the temporarily optimistic type.

**Footnotes**

[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.
Chapter 50

Uncertainty Traps

50.1 Contents

- Overview 50.2
- The Model 50.3
- Implementation 50.4
- Results 50.5
- Exercises 50.6
- Solutions 50.7
- Exercise 2 50.8

50.2 Overview

In this lecture we study a simplified version of an uncertainty traps model of Fajgelbaum, Schaal and Taschereau-Dumouchel [30].

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.

50.3 The Model

The original model described in [30] has many interesting moving parts. Here we examine a simplified version that nonetheless captures many of the key ideas.

50.3.1 Fundamentals

The evolution of the fundamental process \( \{ \theta_t \} \) is given by

\[
\theta_{t+1} = \rho \theta_t + \sigma \theta w_{t+1}
\]

where

- \( \sigma > 0 \) and \( 0 < \rho < 1 \)
- \( \{ w_t \} \) is IID and standard normal

The random variable \( \theta_t \) is not observable at any time.

50.3.2 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

\[
x_m = \theta + \epsilon_m \quad \text{where} \quad \epsilon_m \sim N(0, \gamma_x^{-1})
\]

Here the time subscript has been dropped to simplify notation.

The inverse of the shock variance, \( \gamma_x \), is called the shock’s precision.

The higher is the precision, the more informative \( x_m \) is about the fundamental.

Output shocks are independent across time and firms.

50.3.3 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

- $\mathcal{M} \subset \{1, \ldots, \bar{M} \}$ denote the set of currently active firms
- $M := |\mathcal{M}|$ denote the number of currently active firms
- $X$ be the average output $\frac{1}{M} \sum_{m \in \mathcal{M}} x_m$ of the active firms

With this notation and primes for next period values, we can write the updating of the mean and precision via

$$\mu' = \rho \frac{\mu + M \gamma x X}{\gamma + M \gamma x} \quad (2)$$

$$\gamma' = \left( \frac{\rho^2}{\gamma + M \gamma_x} + \sigma^2_\theta \right)^{-1} \quad (3)$$

These are standard Kalman filtering results applied to the current setting.

Exercise 1 provides more details on how (2) and (3) are derived, and then asks you to fill in remaining steps.

The next figure plots the law of motion for the precision in (3) 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 dif-
different values of $M$.

Thus, if one of these values for $M$ remains fixed, a corresponding steady state is the equilibrium level of precision

- high values of $M$ correspond to greater information about the fundamental, and hence more precision in steady state
- low values of $M$ correspond to less information and more uncertainty in steady state

In practice, as we’ll see, the number of active firms fluctuates stochastically.

### 50.3.4 Participation

Omitting time subscripts once more, entrepreneurs enter the market in the current period if

$$\mathbb{E}[u(x_m - F_m)] > c$$

(4)

Here

- the mathematical expectation of $x_m$ is based on (1) 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).

The utility function has the constant absolute risk aversion form

$$u(x) = \frac{1}{a} (1 - \exp(-ax))$$

(5)

where $a$ is a positive parameter.

Combining (4) and (5), 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

$$\psi(\mu, \gamma, F_m) := \frac{1}{a} \left( 1 - \exp \left( -a\mu + aF_m + \frac{a^2}{2} \left( \frac{1}{\gamma} + \frac{1}{\gamma} \right) \right) \right) - c > 0$$

(6)

### 50.4 Implementation

We want to simulate this economy.

We’ll want a named tuple generator of the kind that we’ve seen before.

And we need methods to update $\theta$, $\mu$ and $\gamma$, as well as to determine the number of active firms and their outputs.
The updating methods follow the laws of motion for $\theta$, $\mu$ and $\gamma$ given above.

The method to evaluate the number of active firms generates $F_1, \ldots, F_M$ and tests condition (6) for each firm.

### 50.4.1 Setup

**In [1]:** using InstantiateFromURL
   # optionally add arguments to force installation: instantiate = true, 
   # precompile = true
   github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

**In [2]:** using LinearAlgebra, Statistics
   using DataFrames, Parameters, Plots
   gr(fmt = :png);

**In [3]:** UncertaintyTrapEcon = @with_kw (a = 1.5, # risk aversion
   $\gamma_X = 0.5$, # production shock precision
   $\varphi = 0.99$, # correlation coefficient for $\theta$
   $\sigma_\theta = 0.5$, # standard dev. of $\theta$ shock
   num_firms = 100, # number of firms
   $\sigma_F = 1.5$, # standard dev. of fixed costs
   c = -420.0, # external opportunity cost
   $\mu_{\text{init}} = 0.0$, # initial value for $\mu$
   $\gamma_{\text{init}} = 4.0$, # initial value for $\gamma$
   $\theta_{\text{init}} = 0.0$, # initial value for $\theta$
   $\sigma_X = \sqrt{a / \gamma_X}$) # standard dev. of shock

**Out[3]:** #NamedTuple_kw#253 (generic function with 2 methods)

In the results below we use this code to simulate time series for the major variables.

### 50.5 Results

Let’s look first at the dynamics of $\mu$, which the agents use to track $\theta$
We see that $\mu$ tracks $\theta$ well when there are sufficient firms in the market.

However, there are times when $\mu$ tracks $\theta$ poorly due to insufficient information.

These are episodes where the uncertainty traps take hold.

During these episodes

- precision is low and uncertainty is high
- few firms are in the market

To get a clearer idea of the dynamics, let’s look at all the main time series at once, for a given set of shocks
Notice how the traps only take hold after a sequence of bad draws for the fundamental.
Thus, the model gives us a propagation mechanism that maps bad random draws into long downturns in economic activity.
50.6 Exercises

50.6.1 Exercise 1

Fill in the details behind (2) and (3) based on the following standard result (see, e.g., p. 24 of [109]).

**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

\[
\mu_0 = \frac{\mu \gamma + M \bar{x} \gamma_x}{\gamma + M \gamma_x} \quad \text{and} \quad \gamma_0 = \gamma + M \gamma_x
\]

50.6.2 Exercise 2

Modulo randomness, replicate the simulation figures shown above

- Use the parameter values listed as defaults in the function UncertaintyTrapEcon.

50.7 Solutions

50.7.1 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.

50.8 Exercise 2

First let’s replicate the plot that illustrates the law of motion for precision, which is
\[ \gamma_{t+1} = \left( \frac{\rho^2}{\gamma_t + M \gamma_x} + \sigma_\theta^2 \right)^{-1} \]

Here \( M \) is the number of active firms. The next figure plots \( \gamma_{t+1} \) against \( \gamma_t \) on a 45 degree diagram for different values of \( M \)

In [4]:

```python
econ = UncertaintyTrapEcon()
@unpack \( \rho, \sigma_\theta, \gamma_x \) = econ  # simplify names

# grid for \( \gamma \) and \( \gamma_{t+1} \)
\( \gamma \) = range(1e-10, 3, length = 200)
M_range = 0:6
\( \gamma_p \) = 1 ./ (\( \rho^2 \) ./ (\( \gamma \) .* \( \gamma_x \) .* M_range') .+ \( \sigma_\theta^2 \))
labels = ["0", "1", "2", "3", "4", "5", "6"]

plot(\( \gamma \), \( \gamma \), lw = 2, label = "45 Degree")
plot!(\( \gamma \), \( \gamma_p \), lw = 2, label = labels)
plot!(xlabel = "Gamma", ylabel = "Gamma", legend_title = "M", legend = :bottomright)
```

Out[4]:

The points where the curves hit the 45 degree lines are the long run steady states corresponding to each \( M \), if that value of \( M \) was to remain fixed. As the number of firms falls, so does the long run steady state of precision.

Next let’s generate time series for beliefs and the aggregates – that is, the number of active firms and average output
In [5]: function simulate(uc, capT = 2_000)
    # unpack parameters
    @unpack a, γ_x, ρ, σ_θ, num_firms, σ_F, c, μ_init, γ_init, θ_init, σ_x = 0
capT

    # draw standard normal shocks
    w_shocks = randn(capT)

    # aggregate functions
    # auxiliary function ψ
    function ψ(γ, μ, F)
        temp1 = -a * (μ - F)
        temp2 = 0.5 * a^2 / (γ + γ_x)
        return (1 - exp(temp1 + temp2)) / a - c
    end

    # compute X, M
    function gen_aggregates(γ, μ, θ)
        F_vals = σ_F * randn(num_firms)
        M = sum(ψ(Ref(γ), Ref(μ), F_vals) ∗ θ) # counts number of active firms
        if any(ψ(γ, μ, f) > 0 for f in F_vals) # an active firm
            x_vals = θ .+ σ_x * randn(M)
            X = mean(x_vals)
        else
            X = 0.0
        end
        return (X = X, M = M)
    end

    # initialize dataframe
    X_init, M_init = gen_aggregates(γ_init, μ_init, θ_init)
    df = DataFrame(γ = γ_init, μ = μ_init, θ = θ_init, X = X_init, M = M_init)

    # update dataframe
    for t in 2:capT
        # unpack old variables
        @unpack θ_old, γ_old, μ_old, X_old, M_old = (df.θ[end], df.γ[end], df.
        μ[end], df.X[end], df.M[end])

        # define new beliefs
        θ = ρ ∗ θ_old + σ_θ ∗ w_shocks[t-1]
        μ = (ρ ∗ (γ_old ∗ μ_old + M_old ∗ γ_x ∗ X_old))/(γ_old + M_old ∗ γ_x)
        γ = 1 / (ρ^2 / (γ_old + M_old ∗ γ_x) + σ_θ^2)

        # compute new aggregates
        X, M = gen_aggregates(γ, μ, θ)
push!(df, (γ = γ, μ = μ, θ = θ, X = X, M = M))
    end

    # return
    return df
end

Out[5]: simulate (generic function with 2 methods)
First let’s see how well $\mu$ tracks $\theta$ in these simulations

In [6]: df = simulate(econ)

    plot(eachindex(df.μ), df.μ, lw = 2, label = "Mu")
    plot!(eachindex(df.θ), df.θ, lw = 2, label = "Theta")
    plot!(xlabel = "x", ylabel = "y", legend_title = "Variable", legend = :bottomright)

Out[6]:

Now let’s plot the whole thing together

In [7]: len = eachindex(df.θ)
    yvals = [df.θ, df.μ, df.γ, df.M]
    vars = ["Theta", "Mu", "Gamma", "M"]

    plt = plot(layout = (4,1), size = (600, 600))
    for i in 1:4
        plot!(plt[i], len, yvals[i], xlabel = "t", ylabel = vars[i], label ="")
    end

    plot(plt)

Out[7]:

---

50.8. EXERCISE 2

Now let’s see how well $\mu$ tracks $\theta$ in these simulations

In [6]: df = simulate(econ)

    plot(eachindex(df.μ), df.μ, lw = 2, label = "Mu")
    plot!(eachindex(df.θ), df.θ, lw = 2, label = "Theta")
    plot!(xlabel = "x", ylabel = "y", legend_title = "Variable", legend = :bottomright)

Out[6]:

Now let’s plot the whole thing together

In [7]: len = eachindex(df.θ)
    yvals = [df.θ, df.μ, df.γ, df.M]
    vars = ["Theta", "Mu", "Gamma", "M"]

    plt = plot(layout = (4,1), size = (600, 600))
    for i in 1:4
        plot!(plt[i], len, yvals[i], xlabel = "t", ylabel = vars[i], label ="")
    end

    plot(plt)

Out[7]:

---
Chapter 51

The Aiyagari Model

51.1 Contents

- Overview 51.2
- The Economy 51.3
- Firms 51.4
- Code 51.5

51.2 Overview

In this lecture we describe the structure of a class of models that build on work by Truman Bewley [12].

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 [1]
- risk sharing and asset pricing [50]
- the shape of the wealth distribution [9]
- etc., etc., etc.

51.2.1 References

The primary reference for this lecture is [1].

A textbook treatment is available in chapter 18 of [68].

A continuous time version of the model by SeHyoun Ahn and Benjamin Moll can be found here.
51.3 The Economy

51.3.1 Households

Infinitely lived households / consumers face idiosyncratic income shocks.

A unit interval of *ex ante* identical households face a common borrowing constraint.

The savings problem faced by a typical household is

\[
\max \mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t)
\]

subject to

\[a_{t+1} + c_t \leq wz_t + (1 + r)a_t \quad c_t \geq 0, \quad \text{and} \quad a_t \geq -B\]

where

- \(c_t\) is current consumption  
- \(a_t\) is assets  
- \(z_t\) is an exogenous component of labor income capturing stochastic unemployment risk, etc.  
- \(w\) is a wage rate  
- \(r\) is a net interest rate  
- \(B\) is the maximum amount that the agent is allowed to borrow

The exogenous process \(\{z_t\}\) follows a finite state Markov chain with given stochastic matrix \(P\).

The wage and interest rate are fixed over time.

In this simple version of the model, households supply labor inelastically because they do not value leisure.

51.4 Firms

Firms produce output by hiring capital and labor.

Firms act competitively and face constant returns to scale.

Since returns to scale are constant the number of firms does not matter.

Hence we can consider a single (but nonetheless competitive) representative firm.

The firm’s output is

\[Y_t = AK_t^\alpha N^{1-\alpha}\]

where

- \(A\) and \(\alpha\) are parameters with \(A > 0\) and \(\alpha \in (0, 1)\)  
- \(K_t\) is aggregate capital  
- \(N\) is total labor supply (which is constant in this simple version of the model)
The firm’s problem is
\[
\max_{K,N} \{ AK_t^\alpha N^{1-\alpha} - (r + \delta)K - wN \}
\]

The parameter $\delta$ is the depreciation rate.

From the first-order condition with respect to capital, the firm’s inverse demand for capital is
\[
r = A \frac{\alpha}{N} \left( \frac{N}{K} \right)^{1-\alpha} - \delta
\]

Using this expression and the firm’s first-order condition for labor, we can pin down the equilibrium wage rate as a function of $r$ as
\[
w(r) = A(1 - \alpha)\frac{A\alpha}{(r + \delta)^{\alpha/(1-\alpha)}}
\]

### 51.4.1 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 (1) and a wage rate $w(r)$ as given in (2)
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.

### 51.5 Code

Let’s look at how we might compute such an equilibrium in practice.

To solve the household’s dynamic programming problem we’ll use the DiscreteDP 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 $\text{DiscreteDP}$.

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 $\text{DiscreteDP}$ 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 object also includes a default set of parameters that we’ll adopt unless otherwise specified.

### 51.5.1 Setup

In [1]: using InstantiateFromURL
   # optionally add arguments to force installation: instantiate = true, []
   ~precompile = true
   github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

In [2]: using LinearAlgebra, Statistics

In [3]: using Parameters, Plots, QuantEcon
   gr(fmt = :png);

In [4]: Household = @with_kw (r = 0.01,
   w = 1.0,
   σ = 1.0,
   β = 0.96,
   z_chain = MarkovChain([0.9 0.1; 0.1 0.9], [0.1; 1.0]),
   a_min = 1e-10,
   a_max = 18.0,
   a_size = 200,
   a_vals = range(a_min, a_max, length = 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),
   u = σ == 1 ? x -> log(x) : x -> (x^(1 - σ) - 1) / (1 - σ),
   # -Inf is the utility of dying (0 consumption)
   R = setup_R!(fill(-Inf, n, a_size), a_vals, s_vals, [],
   # -Inf is the utility of dying (0 consumption)
   Q = setup_Q!(zeros(n, a_size, n), s_i_vals, z_chain))

   function setup_Q!(Q, s_i_vals, z_chain)
     for next_s_i in 1:size(Q, 3)
       for a_i in 1:size(Q, 2)
         for s_i in 1:size(Q, 1)
           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
return Q
end

function setup_R!(R, a_vals, s_vals, r, w, u)
    for new_a_i in 1:size(R, 2)
        a_new = a_vals[new_a_i]
        for s_i in 1:size(R, 1)
            a = s_vals[s_i, 1]
            z = s_vals[s_i, 2]
            c = w * z + (1 + r) * a - a_new
            if c > 0
                R[s_i, new_a_i] = u(c)
            end
        end
    end
end
return R
end

Out[4]: setup_R! (generic function with 1 method)

As a first example of what we can do, let’s compute and plot an optimal accumulation policy at fixed prices.

In [5]: # Create an instance of Household
    am = Household(a_max = 20.0, r = 0.03, w = 0.956)

    # 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
    @unpack z_size, a_size, n, a_vals = am
    z_vals = am.z_chain.state_values

    # 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, 1)

    labels = ["z = \$(z_vals[1])", "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)

Out[5]:
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.

```python
In [6]: # Firms' parameters
const A = 1
const N = 1
const α = 0.33
const β = 0.96
const δ = 0.05

function r_to_w(r)
    return A * (1 - α) * (A * α / (r + δ)) ^ (α / (1 - α))
end

function rd(K)
    return A * α * (N / K) ^ (1 - α) - δ
end

function prices_to_capital_stock(am, r)
    # Set up problem
    w = r_to_w(r)
    @unpack a_vals, s_vals, u = am
    setup_R!(am.R, a_vals, s_vals, r, w, u)
    aiyagari_ddp = DiscreteDP(am.R, am.Q, am.β)

    # 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(β = β, a_max = 20.0)

# Create a grid of r values at which to compute demand and supply of capital
r_vals = range(0.005, 0.04, length = 20)

# Compute supply of capital
k_vals = prices_to_capital_stock.(Ref(am), r_vals)

# Plot against demand for capital by firms
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))
Chapter 52

Default Risk and Income Fluctuations

52.1 Contents

- Overview 52.2
- Structure 52.3
- Equilibrium 52.4
- Computation 52.5
- Results 52.6
- Exercises 52.7
- Solutions 52.8

52.2 Overview

This lecture computes versions of Arellano’s [5] 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.

52.3 Structure

In this section we describe the main features of the model.

52.3.1 Output, Consumption and Debt

A small open economy is endowed with an exogenous stochastically fluctuating potential output stream \( \{y_t\} \).

Potential output is realized only in periods in which the government honors its sovereign debt.

The output good can be traded or consumed.

The sequence \( \{y_t\} \) is described by a Markov process with stochastic density kernel \( p(y, y') \).

Households within the country are identical and rank stochastic consumption streams according to

\[
\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t)
\]

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 (1).

The government is the only domestic actor with access to foreign credit.

Because household are averse to consumption fluctuations, the government will try to smooth consumption by borrowing from (and lending to) foreign creditors.

52.3.2 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

$$c = y + B - q(B', y)B'$$  \hspace{1cm} (2)

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.

### 52.3.3 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

$$q = \frac{1 - \delta}{1 + r}$$  \hspace{1cm} (3)

Next we turn to how the government in effect chooses the default probability $\delta$.

### 52.3.4 Government’s decisions

At each point in time $t$, the government chooses between

1. defaulting
2. meeting its current obligations and purchasing or selling an optimal quantity of one-period sovereign debt
Defaulting means declining to repay all of its current obligations.
If the government defaults in the current period, then consumption equals current output.
But a sovereign default has two consequences:

1. Output immediately falls from $y$ to $h(y)$, where $0 \leq h(y) \leq y$
   - it returns to $y$ only after the country regains access to international credit markets.

1. The country loses access to foreign credit markets.

### 52.3.5 Reentering international credit market

While in a state of default, the economy regains access to foreign credit in each subsequent period with probability $\theta$.

### 52.4 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

1. The interest rate on the government’s debt includes a risk-premium sufficient to make foreign creditors expect on average to earn the constant risk-free international interest rate.

To express these ideas more precisely, consider first the choices of the government, which

1. enters a period with initial assets $B$, or what is the same thing, initial debt to be repaid now of $-B$
2. observes current output $y$, and
3. chooses either
4. to default, or
5. to pay $-B$ and set next period’s debt due to $-B'$

In a recursive formulation,
• state variables for the government comprise the pair \((B, y)\)
• \(v(B, y)\) is the optimum value of the government’s problem when at the beginning of a period it faces the choice of whether to honor or default
• \(v_c(B, y)\) is the value of choosing to pay obligations falling due
• \(v_d(y)\) is the value of choosing to default

\(v_d(y)\) does not depend on \(B\) because, when access to credit is eventually regained, net foreign assets equal 0.

Expressed recursively, the value of defaulting is

\[
v_d(y) = u(h(y)) + \beta \int \{\theta v(0, y') + (1 - \theta)v_d(y')\} p(y, y')dy'
\]

The value of paying is

\[
v_c(B, y) = \max_{B' \geq -Z} \left\{ u(y - q(B', y)B' + B) + \beta \int v(B', y')p(y, y')dy' \right\}
\]

The three value functions are linked by

\[
v(B, y) = \max\{v_c(B, y), v_d(y)\}
\]

The government chooses to default when

\[v_c(B, y) < v_d(y)\]

and hence given \(B'\) the probability of default next period is

\[
\delta(B', y) := \int 1\{v_c(B', y') < v_d(y')\} p(y, y')dy' \tag{4}
\]

Given zero profits for foreign creditors in equilibrium, we can combine (3) and (4) to pin down the bond price function:

\[
q(B', y) = \frac{1 - \delta(B', y)}{1 + r} \tag{5}
\]

52.4.1 Definition of equilibrium

An \emph{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.
CHAPTER 52. DEFAULT RISK AND INCOME FLUCTUATIONS

- 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 (5).

52.5 Computation

Let’s now compute an equilibrium of Arellano’s model.

The equilibrium objects are the value function \( v(B, y) \), the associated default decision rule, and the pricing function \( q(B', y) \).

We’ll use our code to replicate Arellano’s results.

After that we’ll perform some additional simulations.

It uses a slightly modified version of the algorithm recommended by Arellano.

- The appendix to [5] 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 \( v_c(B, y) \)

1. Update the value function \( v(B, y) \), the default rule, the implied ex ante default probability, and the price function.
2. Check for convergence. If converged, stop. If not, go to step 2.

We use simple discretization on a grid of asset holdings and income levels.

The output process is discretized using Tauchen’s quadrature method.

The code can be found below:

(Results and discussion follow the code)

52.5.1 Setup

In [1]: using InstantiateFromURL

# optionally add arguments to force installation: instantiate = true, precompile = true

github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

In [2]: using LinearAlgebra, Statistics

   using Parameters, QuantEcon, DataFrames, Plots, Random
In [3]: function ArellanoEconomy(;β = .953,
γ = 2.,
r = 0.017,
ρ = 0.945,
η = 0.025,
θ = 0.282,
ny = 21,
nB = 251)

# create grids
Bgrid = collect(range(-.4, .4, length = nB))
mc = tauchen(ny, ρ, η)
Π = 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)
vf = zeros(nB, ny)
vc = zeros(nB, ny)
policy = zeros(nB, ny)
q = ones(nB, ny).* (1 / (1 + r))
defprob = zeros(nB, ny)

return (β = β, γ = γ, r = r, ρ = ρ, η = η, θ = θ, ny = ny,
nB = nB, ygrid = ygrid, ydefgrid = ydefgrid,
Bgrid = Bgrid, Π = Π, vf = vf, vd = vd, vc = vc,
policy = policy, q = q, defprob = defprob)
end

u(ae, c) = c^(1 - ae.γ) / (1 - ae.γ)

function one_step_update!(ae,
EV,
EVD,
EVc)

# unpack stuff
@unpack β, γ, r, ρ, η, θ, ny, nB = ae
@unpack ygrid, ydefgrid, Bgrid, Π, vf, vd, vc, policy, q, defprob = ae
zero_ind = searchsortedfirst(Bgrid, 0.)

for iy in 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 in 1:nB
    B = ae.Bgrid[ib]

    current_max = -1e14
    pol_ind = θ
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

# 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

function compute_prices!(ae)
    # unpack parameters
    @unpack β, γ, r, φ, η, θ, ny, nB = ae

    # create default values with a matching size
    vd_compat = repeat(ae.vd, nB)
    default_states = vd_compat .> ae.vc

    # update default probabilities and prices
    copyto!(ae.defprob, default_states * ae.Π)
    copyto!(ae.q, (1 .- ae.defprob) / (1 + r))
    return
end

function vfi!(ae; tol = 1e-8, maxit = 10000)
    # unpack stuff
    @unpack β, γ, r, φ, η, θ, ny, nB = ae
    @unpack ygrid, ydefgrid, Bgrid, Π, vf, vd, vc, policy, q, defprob = ae
    Πt = Π'

    # Iteration stuff
    it = 0
    dist = 10.

    # allocate memory for update
    V_upd = similar(ae.vf)

    while dist > tol && it < maxit
        it += 1

        # compute expectations for this iterations
        # (we need Π' because of order value function dimensions)
        copyto!(V_upd, ae.vf)
        EV = ae.vf * Πt
        EVd = ae.vd * Πt
        EVc = ae.vc * Πt

        # update value function
one_step_update!(ae, EV, EVd, EVc)

# update prices
compute_prices!(ae)

dist = maximum(abs(x - y) for (x, y) in zip(V_upd, ae.vf))

if it % 25 == 0
    println("Finished iteration $it with dist of $(dist)")
end
end

function QuantEcon.simulate(ae, capT = 5000; y_init = mean(ae.ygrid), B_init = mean(ae.Bgrid), )

    # get initial indices
    zero_index = searchsortedfirst(ae.Bgrid, 0.)
    y_init_ind = searchsortedfirst(ae.ygrid, y_init)
    B_init_ind = searchsortedfirst(ae.Bgrid, B_init)

    # 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 = zeros(capT+1)
    B_sim_val, q_sim_val = similar(y_sim_val), similar(y_sim_val)
    B_sim_indices = fill(0, 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 in 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])

            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]]
        end
    end
let q_sim_val[t] = ae.q[B_sim_indices[t + 1], y_sim_indices[t]]
end

# if you are in default
else
    B_sim_indices[t + 1] = zero_index
    B_sim_val[t+1] = 0.
    y_sim_val[t] = ae.ydefgrid[y_sim_indices[t]]
    q_sim_val[t] = ae.q[zero_index, y_sim_indices[t]]

    # with probability θ exit default status
    default_status[t + 1] = rand() ≥ ae.θ
end

return (y_sim_val[1:capT], B_sim_val[1:capT], q_sim_val[1:capT],
        default_status[1:capT])
end

52.6 Results

Let’s start by trying to replicate the results obtained in [5].

In what follows, all results are computed using Arellano’s parameter values.

The values can be seen in the function ArellanoEconomy shown above.

- For example, r=0.017 matches the average quarterly rate on a 5 year US treasury over the period 1983–2001.

Details on how to compute the figures are reported as solutions to the exercises.

The first figure shows the bond price schedule and replicates Figure 3 of Arellano, where \( y_L \) and \( Y_H \) are particular below average and above average values of output \( y \).
• $y_L$ is 5% below the mean of the $y$ grid values
• $y_H$ is 5% above the mean of the $y$ grid values

The grid used to compute this figure was relatively coarse (ny, nB = 21, 251) in order to match Arrelano’s findings.

Here’s the same relationships computed on a finer grid (ny, nB = 51, 551)

In either case, the figure shows that
  • Higher levels of debt (larger $-B’$) induce larger discounts on the face value, which correspond to higher interest rates.
  • Lower income also causes more discounting, as foreign creditors anticipate greater likelihood of default.

The next figure plots value functions and replicates the right hand panel of Figure 4 of [5]
We can use the results of the computation to study the default probability $\delta(B', y)$ defined in (4).

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.

### 52.7 Exercises

#### 52.7.1 Exercise 1

To the extent that you can, replicate the figures shown above

- Use the parameter values listed as defaults in the function ArellanoEconomy.
- The time series will of course vary depending on the shock draws.
52.8 Solutions

In [4]: using DataFrames, Plots
    gr(fmt=:png);

Compute the value function, policy and equilibrium prices

In [5]: ae = ArellanoEconomy(β = .953,  # time discount rate
                                 γ = 2.,  # risk aversion
                                 r = .017,  # international interest rate
                                 p = .945,  # persistence in output
                                 η = 0.025,  # st dev of output shock
                                 θ = 0.282,  # prob of regaining access
                                 ny = 21,  # number of points in y grid
                                 nB = 251)  # number of points in B grid

    # now solve the model on the grid.
    vfi!(ae)

Finished iteration 25 with dist of 0.3424484168091375
Finished iteration 50 with dist of 0.09820394074288075
Finished iteration 75 with dist of 0.02915866229151476
Finished iteration 100 with dist of 0.008729266837651295
Finished iteration 125 with dist of 0.002618400938121823
Finished iteration 150 with dist of 0.0007857709211727126
Finished iteration 175 with dist of 7.078195654131036e-5
Finished iteration 200 with dist of 2.1244388765495614e-5
Finished iteration 225 with dist of 6.376267926100354e-6
Finished iteration 250 with dist of 1.913766855210497e-6
Finished iteration 275 with dist of 5.743961786208729e-7
Finished iteration 300 with dist of 1.723987352875156e-7
Finished iteration 325 with dist of 5.174360495630026e-8
Finished iteration 350 with dist of 1.553028994294364e-8

Compute the bond price schedule as seen in figure 3 of Arellano (2008)

In [6]: # create "Y High" and "Y Low" values as 5% devs from mean
    high, low = 1.05 * mean(ae.ygrid), 0.95 * mean(ae.ygrid)
    iy_high, iy_low = map(x -> searchsortedfirst(ae.ygrid, x), (high, low))

    # extract a suitable plot grid
    x = zeros(0)
    q_low = zeros(0)
    q_high = zeros(0)
    for i in 1:ae.nB
        b = ae.Bgrid[i]
        if -0.35 ≤ b ≤ θ  # to match fig 3 of Arellano
            push!(x, b)
            push!(q_low, ae.q[i, iy_low])
            push!(q_high, ae.q[i, iy_high])
        end
    end

    # generate plot
plot(x, q_low, label = "Low")
plot!(x, q_high, label = "High")
plot!(title = "Bond price schedule q(y, B')",
       xlabel = "B'", ylabel = "q", legend_title = "y", legend = :topleft)

Out[6]:

```
```

Draw a plot of the value functions

In [7]: plot(ae.Bgrid, ae.vf[:, iy_low], label = "Low")
plot!(ae.Bgrid, ae.vf[:, iy_high], label = "High")
plot!(xlabel = "B'", ylabel = "V(y,B)", title = "Value functions",
       legend_title="y", legend = :topleft)

Out[7]:

```
```
Draw a heat map for default probability

```
In [8]: heatmap(ae.Bgrid[1:end-1],
         ae.ygrid[2:end],
         reshape(clamp.(vec(ae.defprob[1:end - 1, 1:end - 1]), 0, 1), 250, 20))
```

```
Out[8]:
```

```
plot!(xlabel = "B'", ylabel = "y", title = "Probability of default",
       legend = :topleft)
```
Plot a time series of major variables simulated from the model

```
In [9]: using Random
   # set random seed for consistent result
Random.seed!(348938)

   # simulate
T = 250
y_vec, B_vec, q_vec, default_vec = simulate(ae, T)

   # find starting and ending periods of recessions
defs = findall(default_vec)
def_breaks = diff(defs) .> 1
def_start = defs[[true; def_breaks]]
def_end = defs[[def_breaks; true]]

y_vals = [y_vec, B_vec, q_vec]
titles = ["Output", "Foreign assets", "Bond price"]

plots = plot(layout = (3, 1), size = (700, 800))

   # Plot the three variables, and for each each variable shading the period(s) of default
   # in grey
   for i in 1:3
       plot!(plots[i], 1:T, y_vals[i], title = titles[i], xlabel = "time",
             label = "", lw = 2)
       for j in 1:length(def_start)
           plot!(plots[i], [def_start[j], def_end[j]],
                 fill(maximum(y_vals[i]), 2),
                 fillrange = [extrema(y_vals[i])...], fcolor = :grey, falpha = 0.3, label = "")
       end
   end

plot(plots)
```

Out[9]:

935
Chapter 53

Globalization and Cycles

53.1 Contents

- Overview 53.2
- Key Ideas 53.3
- Model 53.4
- Simulation 53.5
- Exercises 53.6
- Solutions 53.7

Co-authored with Chase Coleman

53.2 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.

53.2.1 Background

The model builds on work by Judd [60], Deneckner and Judd [23] and Helpman and Krugman [52] 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.

53.3 Key Ideas

It is helpful to begin with an overview of the mechanism.

53.3.1 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.

53.3.2 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.

53.4 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 $t = 0, 1, ...$
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

\[
X_{k,t}^{\frac{1}{\sigma}} = \int_{\Omega_t} [x_{k,t}(\nu)]^{\frac{1}{\sigma}} d\nu
\]

Here \( x_{k,t}(\nu) \) is the total amount of a differentiated good \( \nu \in \Omega_t \) that is produced.

The parameter \( \sigma > 1 \) is the direct partial elasticity of substitution between a pair of varieties and \( \Omega_t \) is the set of varieties available in period \( t \).

We can split the varieties into those which are supplied competitively and those supplied monopolistically; that is, \( \Omega_t = \Omega^c_t + \Omega^m_t \).

### 53.4.1 Prices

Demand for differentiated inputs is

\[
x_{k,t}(\nu) = \left( \frac{p_{k,t}(\nu)}{P_{k,t}} \right)^{-\sigma} \frac{\alpha L_k}{P_{k,t}}
\]

Here

- \( p_{k,t}(\nu) \) is the price of the variety \( \nu \) and
- \( P_{k,t} \) is the price index for differentiated inputs in \( k \), defined by

\[
\left[ P_{k,t} \right]^{1 - \sigma} = \int_{\Omega_t} [p_{k,t}(\nu)]^{1 - \sigma} d\nu
\]

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.

**53.4.2 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
\[ N_{j,t}^m = \theta(M_{j,t} - N_{j,t}^c) \geq 0, \quad \frac{1}{\sigma} \left[ \frac{\alpha L_j}{\theta(M_{j,t} + \rho M_{k,t})} + \frac{\alpha L_k}{\theta(M_{j,t} + M_{k,t}/\rho)} \right] \leq f \]

53.4.3 Law of Motion

With \( \delta \) as the exogenous probability of a variety becoming obsolete, the dynamic equation for the measure of firms becomes

\[ N_{j,t}^c = \delta(N_{j,t}^c + N_{j,t}^m) = \delta(N_{j,t}^c + \theta(M_{j,t} - N_{j,t}^c)) \]

We will work with a normalized measure of varieties

\[ n_{j,t} := \frac{\theta \sigma f N_{j,t}^c}{\alpha (L_1 + L_2)}, \quad i_{j,t} := \frac{\theta \sigma f N_{j,t}^m}{\alpha (L_1 + L_2)}, \quad m_{j,t} := \frac{\theta \sigma f M_{j,t}}{\alpha (L_1 + L_2)} = n_{j,t} + \frac{i_{j,t}}{\theta} \]

We also use \( s_j := \frac{L_j}{L_1 + L_2} \) to be the share of labor employed in country \( j \).

We can use these definitions and the preceding expressions to obtain a law of motion for \( n_t := (n_{1,t}, n_{2,t}) \).

In particular, given an initial condition, \( n_0 = (n_{1,0}, n_{2,0}) \in \mathbb{R}_+^2 \), the equilibrium trajectory, \( \{n_t\}_{t=0}^\infty = \{(n_{1,t}, n_{2,t})\}_{t=0}^\infty \), is obtained by iterating on \( n_{t+1} = F(n_t) \) where \( F : \mathbb{R}_+^2 \rightarrow \mathbb{R}_+^2 \) is given by

\[
F(n_t) = \begin{cases} 
(\delta(\theta s_1 (\rho) + (1 - \theta)n_{1,t}), \delta(\theta s_2 (\rho) + (1 - \theta)n_{2,t})) & \text{for } n_t \in D_{LL} \\
(\delta n_{1,t}, \delta n_{2,t}) & \text{for } n_t \in D_{HH} \\
(\delta n_{1,t}, \delta(h_2(n_{1,t}) + (1 - \theta)n_{2,t})) & \text{for } n_t \in D_{HL} \\
(\delta(h_1(n_{2,t}) + (1 - \theta)n_{1,t}, \delta n_{2,t})) & \text{for } n_t \in D_{LH}
\end{cases}
\]

Here

\[
D_{LL} := \{(n_1, n_2) \in \mathbb{R}_+^2 | n_1 \leq s_j(\rho)\} \\
D_{HH} := \{(n_1, n_2) \in \mathbb{R}_+^2 | n_1 \geq h_j(\rho)\} \\
D_{HL} := \{(n_1, n_2) \in \mathbb{R}_+^2 | n_1 \geq s_1(\rho) \text{ and } n_2 \leq h_2(n_1)\} \\
D_{LH} := \{(n_1, n_2) \in \mathbb{R}_+^2 | n_1 \leq h_1(n_2) \text{ and } n_2 \geq s_2(\rho)\}
\]

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

$$h_j(n_k)^2 + \left((\rho + \frac{1}{\rho})n_k - s_j - s_k\right) h_j(n_k) + \left(n_k^2 - \frac{s_jn_k}{\rho} - s_kn_k\rho\right) = 0$$

### 53.5 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.

#### 53.5.1 Setup

```julia
In [1]: using InstantiateFromURL
   # optionally add arguments to force installation: instantiate = true, precompile = true
github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

In [2]: using LinearAlgebra, Statistics
   using Plots, Parameters
   gr(fmt = :png);

In [3]: function h_j(j, nk, s1, s2, θ, δ, ρ)
   # 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 = (nk * nk - (s_j * nk) / ρ - sk * ρ * nk)
   # Positive solution of quadratic form
   root = (-b + sqrt(b * b - 4 * a * c)) / (2 * a)
   return root
end

DLL(n1, n2, s1_ρ, s2_ρ, s1, s2, θ, δ, ρ) =
```
(n1 ≤ s1_ρ) && (n2 ≤ s2_ρ)

DHH(n1, n2, s1_ρ, s2_ρ, s1, s2, θ, δ, ρ) =
(n1 ≥ h_j(1, n2, s1, s2, θ, δ, ρ)) && (n2 ≥ h_j(2, n1, s1, s2, θ, δ, ρ))

DHL(n1, n2, s1_ρ, s2_ρ, s1, s2, θ, δ, ρ) =
(n1 ≥ h_j(1, n2, s1, s2, θ, δ, ρ)) && (n2 ≤ h_j(2, n1, s1, s2, θ, δ, ρ))

DLH(n1, n2, s1_ρ, s2_ρ, s1, s2, θ, δ, ρ) =
(n1 ≤ h_j(1, n2, s1, s2, θ, δ, ρ)) && (n2 ≥ s2_ρ)

function one_step(n1, n2, s1_ρ, s2_ρ, s1, s2, θ, δ, ρ)
# 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)
elseif DHH(n1, n2, s1_ρ, s2_ρ, s1, s2, θ, δ, ρ)
    n1_tp1 = δ * n1
    n2_tp1 = δ * n2
elseif DHL(n1, n2, s1_ρ, s2_ρ, s1, s2, θ, δ, ρ)
    n1_tp1 = δ * (θ * h_j(2, n1, s1, s2, θ, δ, ρ) + (1 - θ) * n1)
    n2_tp1 = δ * n2
else
    n1_tp1 = δ * (θ * h_j(1, n2, s1, s2, θ, δ, ρ) + (1 - θ) * n2)
    n2_tp1 = δ * n2
end

return n1_tp1, n2_tp1
end

new_n1n2(n1_0, n2_0, s1_ρ, s2_ρ, s1, s2, θ, δ, ρ) =
one_step(n1_0, n2_0, s1_ρ, s2_ρ, s1, s2, θ, δ, ρ)

function pers_till_sync(n1_0, n2_0, s1_ρ, s2_ρ, s1, s2, θ, δ, ρ, maxiter, npers)
# 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

```plaintext
defined 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(s1_ρ, s2_ρ, s1, s2, θ, δ, ρ, maxiter, npers, npts)
  # Create unit range with npts
  synchronized, pers_2_sync = false, 0
  unit_range = range(0.0, 1.0, length = npts)

  # Allocate space to store time to sync
  time_2_sync = zeros(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

  return time_2_sync
end

# model
function MSGSync(s1 = 0.5, θ = 2.5, δ = 0.7, ρ = 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

  return (s1 = s1, s2 = s2, s1_ρ = s1_ρ, s2_ρ = s2_ρ, θ = θ, δ = δ, ρ = ρ)
end

function simulate_n(model, n1_0, n2_0, T)
  # Unpack parameters
  @unpack s1, s2, θ, δ, ρ, s1_ρ, s2_ρ = model

  # Allocate space
  n1 = zeros(T)
  n2 = zeros(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_n1n2(n1_0, n2_0, s1_ρ, s2_ρ, s1, s2, θ, δ, ρ)
  end

  return n1, n2
end
function pers_till_sync(model, n1_0, n2_0,
  maxiter = 500, npers = 3)
  # Unpack parameters
  @unpack s1, s2, θ, δ, ρ, s1_ρ, s2_ρ = 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;
  maxiter = 250,
  npers = 3,
  npts = 50)
  # Unpack parameters
  @unpack s1, s2, θ, δ, ρ, s1_ρ, s2_ρ = model
  ab = create_attraction_basis(s1_ρ, s2_ρ, s1, s2, θ, δ,
    ρ, maxiter, npers, npts)
  return ab
end

Out[3]: create_attraction_basis (generic function with 2 methods)

53.5.2 Time Series of Firm Measures

We write a short function below that exploits the preceding code and plots two time series.
Each time series gives the dynamics for the two countries.
The time series share parameters but differ in their initial condition.
Here’s the function

In [4]: function plot_timeseries(n1_0, n2_0, s1 = 0.5, θ = 2.5, δ = 0.7, ρ = 0.2)
  model = MSGSync(s1, θ, δ, ρ)
  n1, n2 = simulate_n(model, n1_0, n2_0, 25)
  return [n1 n2]
end

# Create figures
data_ns = plot_timeseries(0.15, 0.35)
data_s = plot_timeseries(0.4, 0.3)
plot(data_ns, title = "Not Synchronized", legend = false)

Out[4]:
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.
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.

53.6 Exercises

53.6.1 Exercise 1

Replicate the figure shown above by coloring initial conditions according to whether or not synchronization occurs from those conditions.
53.7 Solutions

53.7.1 Exercise 1

In [6]: function plot_attraction_basis(s1 = 0.5, θ = 2.5, δ = 0.7, ρ = 0.2; npts = 250)
   # Create attraction basis
   unitrange = range(0, 1, length = npts)
   model = MSGSync(s1, θ, δ, ρ)
   ab = create_attraction_basis(model, npts=npts)
   plt = Plots.heatmap(ab, legend = false)
   end

Out[6]: plot_attraction_basis (generic function with 5 methods)

In [7]: 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]]
   plots = (plot_attraction_basis(p...) for p in params)
   plot(plots..., size = (1000, 1000))

Out[7]:
Part VII

Time Series Models
Chapter 54

Covariance Stationary Processes

54.1 Contents

- Overview 54.2
- Introduction 54.3
- Spectral Analysis 54.4
- Implementation 54.5

54.2 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.

54.2.1 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.

54.2.2 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.

### 54.2.3 Other Reading

For supplementary reading, see.

- [68], chapter 2
- [94], chapter 11
- John Cochrane’s notes on time series analysis, chapter 8
- [98], chapter 6
- [18], all

### 54.2.4 Setup

In [1]: `using InstantiateFromURL
# optionally add arguments to force installation: instantiate = true, precompile = true
github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")`

In [2]: `using LinearAlgebra, Statistics`

### 54.3 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).

### 54.3.1 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 \( t \).
54.3. INTRODUCTION

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.

54.3.2 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.

54.3.3 Example 2: General Linear Processes

From the simple building block provided by white noise, we can construct a very flexible family of covariance stationary processes — the general linear processes

$$X_t = \sum_{j=0}^{\infty} \psi_j \epsilon_{t-j}, \quad t \in \mathbb{Z} \tag{1}$$

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 (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 (1) is

$$\gamma(k) = \sigma^2 \sum_{j=0}^{\infty} \psi_j \psi_{j+k} \tag{2}$$

By the Cauchy-Schwartz inequality one can show that $\gamma(k)$ satisfies equation (2).

Evidently, $\gamma(k)$ does not depend on $t$. 
54.3.4 Wold’s Decomposition

Remarkably, the class of general linear processes goes a long way towards describing the entire class of zero-mean covariance stationary processes.

In particular, Wold’s decomposition theorem states that every zero-mean covariance stationary process \( \{X_t\} \) can be written as

\[
X_t = \sum_{j=0}^{\infty} \psi_j \epsilon_{t-j} + \eta_t
\]

where

- \( \{\epsilon_t\} \) is white noise
- \( \{\psi_t\} \) is square summable
- \( \eta_t \) 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 [94], p. 286.

54.3.5 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 (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 \( \sigma = 1 \)

In [3]: using Plots
   gr(fmt=:png);
   plt_1=plot()
   plt_2=plot()
   plots = [plt_1, plt_2]
   for (i, \phi) in enumerate((0.8, -0.8))
       times = 0:0.16
       acov = [\phi^k . / (1 - \phi.^2) for k in times]
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

### 54.3.6 ARMA Processes

A stochastic process \( \{X_t\} \) is called an autoregressive moving average process, or ARMA(p,q), if it can be written as
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) as

\[
L^0 X_t = \phi_1 L^1 X_t - \phi_p L^p X_t = L^0 \epsilon_t + \theta_1 L^1 \epsilon_t + \cdots + \theta_q L^q \epsilon_t \tag{6}
\]

If we let \( \phi(z) \) and \( \theta(z) \) be the polynomials

\[
\phi(z) := 1 - \phi_1 z - \cdots - \phi_p z^p \quad \text{and} \quad \theta(z) := 1 + \theta_1 z + \cdots + \theta_q z^q \tag{7}
\]

then (6) becomes

\[
\phi(L) X_t = \theta(L) \epsilon_t \tag{8}
\]

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 [18].

The function \( t \mapsto \psi_t \) is often called the **impulse response function**.

### 54.4 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.
54.4.1 Complex Numbers

Before discussing the spectral density, we invite you to recall the main properties of complex numbers (or skip to the next section).

It can be helpful to remember that, in a formal sense, complex numbers are just points \((x, y) \in \mathbb{R}^2\) endowed with a specific notion of multiplication.

When \((x, y)\) is regarded as a complex number, \(x\) is called the real part and \(y\) is called the imaginary part.

The modulus or absolute value of a complex number \(z = (x, y)\) is just its Euclidean norm in \(\mathbb{R}^2\), but is usually written as \(|z|\) instead of \(\|z\|\).

The product of two complex numbers \((x, y)\) and \((u, v)\) is defined to be \((xu - vy, xv + yu)\), while addition is standard pointwise vector addition.

When endowed with these notions of multiplication and addition, the set of complex numbers forms a field — addition and multiplication play well together, just as they do in \(\mathbb{R}\).

The complex number \((x, y)\) is often written as \(x + iy\), where \(i\) is called the imaginary unit, and is understood to obey \(i^2 = -1\).

The \(x + iy\) notation provides an easy way to remember the definition of multiplication given above, because, proceeding naively,

\[(x + iy)(u + iv) = xu - yv + i(xv + yu)\]

Converted back to our first notation, this becomes \((xu - vy, xv + yu)\) as promised.

Complex numbers can be represented in the polar form \(re^{i\omega}\) where

\[re^{i\omega} := r(\cos(\omega) + i\sin(\omega)) = x + iy\]

where \(x = r\cos(\omega), y = r\sin(\omega),\) and \(\omega = \arctan(y/x)\) or \(\tan(\omega) = y/x\).

54.4.2 Spectral Densities

Let \(\{X_t\}\) be a covariance stationary process with autocovariance function \(\gamma\) satisfying \(\sum_k \gamma(k)^2 < \infty\).

The spectral density \(f\) of \(\{X_t\}\) is defined as the discrete time Fourier transform of its autocovariance function \(\gamma\)

\[f(\omega) := \sum_{k \in \mathbb{Z}} \gamma(k)e^{-i\omega k}, \quad \omega \in \mathbb{R}\]

(Some authors normalize the expression on the right by constants such as \(1/\pi\) — the convention chosen makes little difference provided you are consistent)

Using the fact that \(\gamma\) is even, in the sense that \(\gamma(t) = \gamma(-t)\) for all \(t\), we can show that

\[f(\omega) = \gamma(0) + 2 \sum_{k \geq 1} \gamma(k) \cos(\omega k)\] \hspace{1cm} (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 \([0, \pi]\).

### 54.4.3 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”)

### 54.4.4 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

\[
f(\omega) = \sigma^2 (1 + 2\theta \cos(\omega) + \theta^2)
\]

With a bit more effort, it is possible to show (see, e.g., p. 261 of [94]) that the spectral density of the AR(1) process \( X_t = \phi X_{t-1} + \epsilon_t \) is

\[
f(\omega) = \frac{\sigma^2}{1 - 2\phi \cos(\omega) + \phi^2}
\]

More generally, it can be shown that the spectral density of the ARMA process (5) is

\[
f(\omega) = \left| \frac{\theta(e^{i\omega})}{\phi(e^{i\omega})} \right|^2 \sigma^2
\]

where

- \( \sigma \) is the standard deviation of the white noise process \( \{\epsilon_t\} \)
- the polynomials \( \phi(\cdot) \) and \( \theta(\cdot) \) are as defined in (7)

The derivation of (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, [94], chapter 11, section 4.

It is a nice exercise to verify that (10) and (11) are indeed special cases of (12).

### 54.4.5 Interpreting the Spectral Density

Plotting (11) reveals the shape of the spectral density for the AR(1) model when \( \phi \) takes the values 0.8 and -0.8 respectively
These spectral densities correspond to the autocovariance functions for the AR(1) process shown above.

Informally, we think of the spectral density as being large at those $\omega \in [0, \pi]$ 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

$$f(\omega) = \gamma(0) + 2 \sum_{k \geq 1} \gamma(k) \cos(\omega k) = \gamma(0) + 2 \sum_{k \geq 1} (-0.8)^k \cos(\omega k)$$

(13)
When we evaluate this at \( \omega = \pi \), we get a large number because \( \cos(\pi k) \) is large and positive when \((-0.8)^k\) is positive, and large in absolute value and negative when \((-0.8)^k\) is negative.

Hence the product is always large and positive, and hence the sum of the products on the right-hand side of (13) is large.

These ideas are illustrated in the next figure, which has \( k \) on the horizontal axis.

In [5]: \( \phi = -0.8 \)

\[
\begin{align*}
\text{times} & = 0:16 \\
y1 & = [\phi^*k ./ (1 - \phi.*2) \text{ for } k \text{ in times}] \\
y2 & = [\cos(\pi * k) \text{ for } k \text{ in times}] \\
y3 & = [a * b \text{ for } (a, b) \text{ in zip}(y1, y2)]
\end{align*}
\]

# Autocovariance when \( \chi = -0.8 \)

\[
\text{plt}_1 = \text{plot}(\text{times}, y1, \text{color}=\text{blue, lw}=2, \text{marker}=\text{circle, markersize}=3, \text{alpha}=0.6, \text{label}="\text{gamma(k)}")}
\]

\[
\text{plot!(plt}_1, \text{seriestype}=\text{hline, [0], linestyle}=\text{dash, alpha}=0.5, \text{lw}=2, \text{label}=""")}
\]

\[
\text{plot!(plt}_1, \text{legend}=\text{topright, xlim}=(0,15), \text{yticks}=[-2, 0, 2])
\]

# Cycles at frequency \( \pi \)

\[
\text{plt}_2 = \text{plot}(\text{times}, y2, \text{color}=\text{blue, lw}=2, \text{marker}=\text{circle, markersize}=3, \text{alpha}=0.6, \text{label}="\text{cos(pi k)}")}
\]

\[
\text{plot!(plt}_2, \text{seriestype}=\text{hline, [0], linestyle}=\text{dash, alpha}=0.5, \text{lw}=2, \text{label}=""")}
\]

\[
\text{plot!(plt}_2, \text{legend}=\text{topright, xlim}=(0,15), \text{yticks}=[-1, 0, 1])
\]

# Product

\[
\text{plt}_3 = \text{plot}(\text{times}, y3, \text{seriestype}=\text{sticks, marker}=\text{circle, markersize}=3, \text{lw}=2, \text{label}="\text{gamma(k) cos(pi k)}")}
\]

\[
\text{plot!(plt}_3, \text{seriestype}=\text{hline, [0], linestyle}=\text{dash, alpha}=0.5, \text{lw}=2, \text{label}=""")}
\]

\[
\text{plot!(plt}_3, \text{legend}=\text{topright, xlim}=(0,15), \text{ylim}=(-3,3), \text{yticks}=[-1, 0, 1, 2, 3])
\]

\[
\text{plot(plt}_1, \text{plt}_2, \text{plt}_3, \text{layout}=(3,1), \text{size}=(800,600))
\]

Out[5]:
On the other hand, if we evaluate $f(\omega)$ at $\omega = \pi/3$, then the cycles are not matched, the sequence $\gamma(k) \cos(\omega k)$ contains both positive and negative terms, and hence the sum of these terms is much smaller.

In [6]:
\[
\phi = -0.8
\]
\[
times = 0:16
\]
\[
y1 = [\phi.^k ./ (1 - \phi.^2) \text{ for } k \text{ in times}]
\]
\[
y2 = [\cos(\pi * k/3) \text{ for } k \text{ in times}]
\]
\[
y3 = [a * b \text{ for } (a, b) \text{ in zip(y1, y2)]}
\]

# Autocovariance when $\phi = -0.8$
\[
\text{plt}_1 = \text{plot(times, y1, color=blue, lw=2, marker=circle, markersize=3, alpha=0.6, label="\gamma(k)"})
\]
\[
\text{plt}(\text{plt}_1, \text{seriestyle=hline, [0], linestyle=dash, alpha=0.5, lw=2, label=""})
\]
\[
\text{plt}(\text{plt}_1, \text{legend=topright, xlim=(0,15), yticks=[-2, 0, 2]})
\]

# Cycles at frequency $\pi$
\[
\text{plt}_2 = \text{plot(times, y2, color=blue, lw=2, marker=circle, markersize=3, alpha=0.6, label="\cos(\pi k/3)"})
\]
\[
\text{plt}(\text{plt}_2, \text{seriestyle=hline, [0], linestyle=dash, alpha=0.5, lw=2, label=""})
\]
\[
\text{plt}(\text{plt}_2, \text{legend=topright, xlim=(0,15), yticks=[-1, 0, 1]})
\]

# Product
\[
\text{plt}_3 = \text{plot(times, y3, seriestyle=sticks, marker=circle, markersize=3, lw=2, label="\gamma(k) \cos(\pi k/3)"})
\]
\[
\text{plt}(\text{plt}_3, \text{seriestyle=hline, [0], linestyle=dash, alpha=0.5, lw=2, label=""})
\]
\[
\text{plt}(\text{plt}_3, \text{legend=topright, xlim=(0,15), ylim=(-3,3), yticks=[-1, 0, 1, 2, 3]})
\]
In summary, the spectral density is large at frequencies $\omega$ where the autocovariance function exhibits damped cycles.

### 54.4.6 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

$$\gamma(k) = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(\omega) e^{i\omega k} d\omega$$  \hspace{1cm} (14)
This is convenient in situations where the spectral density is easier to calculate and manipulate than the autocovariance function.

(For example, the expression (12) for the ARMA spectral density is much easier to work with than the expression for the ARMA autocovariance)

### 54.4.7 Mathematical Theory

This section is loosely based on [94], p. 249-253, and included for those who

- would like a bit more insight into spectral densities
- and have at least some background in Hilbert space theory

Others should feel free to skip to the next section — none of this material is necessary to progress to computation.

Recall that every separable Hilbert space $H$ has a countable orthonormal basis $\{h_k\}$.

The nice thing about such a basis is that every $f \in H$ satisfies

$$f = \sum_k \alpha_k h_k \quad \text{where} \quad \alpha_k := \langle f, h_k \rangle$$

(15)

where $\langle \cdot, \cdot \rangle$ denotes the inner product in $H$.

Thus, $f$ can be represented to any degree of precision by linearly combining basis vectors.

The scalar sequence $\alpha = \{\alpha_k\}$ is called the Fourier coefficients of $f$, and satisfies $\sum_k |\alpha_k|^2 < \infty$.

In other words, $\alpha$ is in $\ell_2$, the set of square summable sequences.

Consider an operator $T$ that maps $\alpha \in \ell_2$ into its expansion $\sum_k \alpha_k h_k \in H$.

The Fourier coefficients of $T\alpha$ are just $\alpha = \{\alpha_k\}$, as you can verify by confirming that $\langle T\alpha, h_k \rangle = \alpha_k$.

Using elementary results from Hilbert space theory, it can be shown that

- $T$ is one-to-one — if $\alpha$ and $\beta$ are distinct in $\ell_2$, then so are their expansions in $H$
- $T$ is onto — if $f \in H$ then its preimage in $\ell_2$ is the sequence $\alpha$ given by $\alpha_k = \langle f, h_k \rangle$
- $T$ is a linear isometry — in particular $\langle \alpha, \beta \rangle = \langle T\alpha, T\beta \rangle$

Summarizing these results, we say that any separable Hilbert space is isometrically isomorphic to $\ell_2$.

In essence, this says that each separable Hilbert space we consider is just a different way of looking at the fundamental space $\ell_2$.

With this in mind, let’s specialize to a setting where

- $\gamma \in \ell_2$ is the autocovariance function of a covariance stationary process, and $f$ is the spectral density
- $H = L_2$, where $L_2$ is the set of square summable functions on the interval $[-\pi, \pi]$, with inner product $\langle g, h \rangle = \int_{-\pi}^{\pi} g(\omega) h(\omega) d\omega$
- $\{h_k\}$ is the orthonormal basis for $L_2$ given by the set of trigonometric functions

$$h_k(\omega) = e^{i\omega k} \sqrt{\frac{2}{\pi}}, \quad k \in \mathbb{Z}, \quad \omega \in [-\pi, \pi]$$
Using the definition of $T$ from above and the fact that $f$ is even, we now have

$$T\gamma = \sum_{k \in \mathbb{Z}} \gamma(k) e^{i\omega k} \sqrt{2\pi} = \frac{1}{\sqrt{2\pi}} f(\omega)$$  \hspace{1cm} (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 (16) gives (14), justifying our earlier expression for the inverse transform.

### 54.5 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

#### 54.5.1 Application

Let’s use this code to replicate the plots on pages 68–69 of [68].

Here are some functions to generate the plots

In [7]: using QuantEcon, Random

```julia
# plot functions
function plot_spectral_density(arma, plt)
    (w, spect) = spectral_density(arma, two_pi=false)
    plot!(plt, w, spect, lw=2, alpha=0.7, label="")
    plot!(plt, title="Spectral density", xlim=(0,\pi),
          xlabel="Frequency", ylabel="Spectrum", yscale=:log)
    return plt
end

function plot_spectral_density(arma)
    plt = plot()
```
function plot_autocovariance(arma, plt)
    acov = autocovariance(arma)
    n = length(acov)
    plot!(plt, 0:(n-1), acov, seriestype=:sticks, marker=:circle,
          markersize=2, label="")
    plot!(plt, seriestype=:hline, [0], color=:red, label="")
    plot!(plt, title="Autocovariance", xlim=(-0.5, n-0.5),
          xlabel="time", ylabel="autocovariance")
    return plt
end

function plot_impulse_response(arma, plt)
    psi = impulse_response(arma)
    n = length(psi)
    plot!(plt, 0:(n-1), psi, seriestype=:sticks, marker=:circle,
          markersize=2, label="")
    plot!(plt, seriestype=:hline, [0], color=:red, label="")
    plot!(plt, title="Impulse response", xlim=(-0.5,n-0.5),
          xlabel="time", ylabel="response")
    return plt
end

function plot_simulation(arma, plt)
    X = simulation(arma)
    n = length(X)
    plot!(plt, 0:(n-1), X, lw=2, alpha=0.7, label="")
    plot!(plt, title="Sample path", xlim=(0,0,n), xlabel="time",
          ylabel="state space")
    return plt
end

function quad_plot(arma)
    plt_1 = plot()
    plt_2 = plot()
    plt_3 = plot()
    plt_4 = plot()
plots = [plt_1, plt_2, plt_3, plt_4]

plot_functions = [plot_spectral_density,
                 plot_impulse_response,
                 plot_autocovariance,
                 plot_simulation]

for (i, plt, plot_func) in zip(1:1:4, plots, plot_functions)
    plots[i] = plot_func(arma, plt)
end

return plot(plots[1], plots[2], plots[3], plots[4], layout=(2,2), size=(800,800))
end

Out[7]: quad_plot (generic function with 1 method)

Now let’s call these functions to generate the plots. We’ll use the model \( X_t = 0.5X_{t-1} + \epsilon_t - 0.8\epsilon_{t-2} \)

In [8]: Random.seed!(42) # For reproducible results.
   \[ \phi = 0.5; \]
   \[ \theta = [0, -0.8]; \]
   \[ arma = ARMA(\phi, \theta, 1.0) \]
   \[ quad_plot(arma) \]

Out[8]:
54.5. IMPLEMENTATION

54.5.2 Explanation

The call

```
arma = ARMA(ϕ, θ, σ)
```

creates an instance `arma` that represents the ARMA($p$, $q$) model

\[
X_t = \phi_1 X_{t-1} + \cdots + \phi_p X_{t-p} + \epsilon_t + \theta_1 \epsilon_{t-1} + \cdots + \theta_q \epsilon_{t-q}
\]

If $\phi$ and $\theta$ are arrays or sequences, then the interpretation will be

- $\phi$ 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.

54.5.3 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

$$a_k = \frac{1}{n} \sum_{t=0}^{n-1} A_t e^{i k 2 \pi t / n}$$

Thus, if we set $A_t = f(\omega_t)$, where $f$ is the spectral density and $\omega_t := 2 \pi t / n$, then

$$a_k = \frac{1}{n} \sum_{t=0}^{n-1} f(\omega_t) e^{i \omega_t k} = \frac{1}{2 \pi} \frac{1}{n} \sum_{t=0}^{n-1} f(\omega_t) e^{i \omega_t k}, \quad \omega_t := 2 \pi t / n$$

For $n$ sufficiently large, we then have

$$a_k \approx \frac{1}{2 \pi} \int_0^{2 \pi} f(\omega) e^{i \omega k} d\omega = \frac{1}{2 \pi} \int_{-\pi}^{\pi} f(\omega) e^{i \omega k} d\omega$$

(You can check the last equality)

In view of (14) we have now shown that, for $n$ sufficiently large, $a_k \approx \gamma(k)$ — which is exactly what we want to compute.
Chapter 55

Estimation of Spectra

55.1 Contents

- Overview 55.2
- Periodograms 55.3
- Smoothing 55.4
- Exercises 55.5
- Solutions 55.6

55.2 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 [94] or [18].

55.2.1 Setup

In [1]: using InstantiateFromURL
   # optionally add arguments to force installation: instantiate = true,
   # precompile = true
   github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

In [2]: using LinearAlgebra, Statistics
55.3 Periodograms

Recall that the spectral density $f$ of a covariance stationary process with autocorrelation function $\gamma$ can be written

$$f(\omega) = \gamma(0) + 2 \sum_{k \geq 1} \gamma(k) \cos(\omega k), \quad \omega \in \mathbb{R}$$

Now consider the problem of estimating the spectral density of a given time series, when $\gamma$ is unknown.

In particular, let $X_0, \ldots, X_{n-1}$ be $n$ consecutive observations of a single time series that is assumed to be covariance stationary.

The most common estimator of the spectral density of this process is the periodogram of $X_0, \ldots, X_{n-1}$, which is defined as

$$I(\omega) := \frac{1}{n} \left| \sum_{t=0}^{n-1} X_t e^{i t \omega} \right|^2, \quad \omega \in \mathbb{R} \quad (1)$$

(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.

55.3.1 Interpretation

To interpret the periodogram, it is convenient to focus on its values at the Fourier frequencies

$$\omega_j := \frac{2\pi j}{n}, \quad j = 0, \ldots, n - 1$$

In what sense is $I(\omega_j)$ an estimate of $f(\omega_j)$?

The answer is straightforward, although it does involve some algebra.

With a bit of effort one can show that, for any integer $j > 0$,

$$\sum_{t=0}^{n-1} e^{i t \omega_j} = \sum_{t=0}^{n-1} \exp \left\{ i 2\pi j \frac{t}{n} \right\} = 0$$
55.3. PERIODOGRAMS

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 $f(\omega_j)$.

55.3.2 Calculation

Let’s now consider how to compute the periodogram as defined in (1).

There are already functions available that will do this for us — an example is periodogram in the DSP.jl package.

However, it is very simple to replicate their results, and this will give us a platform to make useful extensions.

The most common way to calculate the periodogram is via the discrete Fourier transform, which in turn is implemented through the fast Fourier transform algorithm.

In general, given a sequence $a_0, \ldots, a_{n-1}$, 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
can be computed by `abs(fft(X)).^2 / length(X)`.

Note: The Julia function `abs` acts elementwise, and correctly handles complex numbers (by computing their modulus, which is exactly what we need).

A function called `periodogram` that puts all this together can be found [here](#).

Let’s generate some data for this function using the `ARMA` type from `QuantEcon.jl` (see the lecture on linear processes for more details).

Here’s a code snippet that, once the preceding code has been run, generates data from the process

\[
X_t = 0.5X_{t-1} + \epsilon_t - 0.8\epsilon_{t-2}
\]  

where \(\{\epsilon_t\}\) is white noise with unit variance, and compares the periodogram to the actual spectral density

In [3]: using QuantEcon, Plots, Random

```julia
gr(fmt = :png);
Random.seed!(42) # For reproducible results.

n = 40 # Data size
ϕ = 0.5 # AR parameter
θ = [0, -0.8] # MA parameter
σ = 1.0
lp = ARMA(ϕ, θ, σ)
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")
```

Out[3]:
This estimate looks rather disappointing, but the data size is only 40, so perhaps it’s not surprising that the estimate is poor.

However, if we try again with $n = 1200$ the outcome is not much better.

The periodogram is far too irregular relative to the underlying spectral density.

This brings us to our next topic.
55.4 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

$$I_S(\omega_j) := \sum_{\ell=-p}^{p} w(\ell) I(\omega_{j+\ell}) \tag{3}$$

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 (3)

In [4]: 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 = range(-12, 12, length = 25)
plot(x, window, color="darkblue", title="Hanning window", ylabel="Weights",
     xlabel="Position in sequence of weights", legend=false, grid=false)
55.4. **SMOOTHING**

55.4.1 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()`, and `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 (2) — and there are 400 observations)
From top figure to bottom, the window length is varied from small to large.

In looking at the figure, we can see that for this model and data size, the window length chosen in the middle figure provides the best fit.

Relative to this value, the first window length provides insufficient smoothing, while the third gives too much smoothing.

Of course in real estimation problems the true spectral density is not visible and the choice of appropriate smoothing will have to be made based on judgement/priors or some other theory.

55.4.2 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 (3).

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.

55.4.3 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

\[
X_{t+1} = \mu + \phi X_t + \epsilon_{t+1}
\]

(4)

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
\[ f(\omega) = \left| \frac{1}{1 - \phi e^{i\omega}} \right|^2 g(\omega) \]  

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 \texttt{ar_periodogram()} — the third function in \texttt{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 \texttt{ar_periodogram()}

The periodograms are calculated from time series drawn from (4) 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.

55.5 Exercises

55.5.1 Exercise 1

Replicate this figure (modulo randomness).

The model is as in equation (2) and there are 400 observations.

For the smoothed periodogram, the window type is “hamming”.
55.5.2 Exercise 2

Replicate this figure (modulo randomness).

The model is as in equation (4), 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.

55.6 Solutions

55.6.1 Exercise 1

In [5]: 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)

    t = "window length = $wl"
    push!(titles, t)
end

In [6]: 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)))

Out[6]:

55.6. SOLUTIONS

55.6.2 Exercise 2

In [7]: 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,
          label="spectral density", legend=:topleft)

    x, y_smoothed = periodogram(X, "hamming", w1)
    plot!(p[i], x, y_smoothed, linecolor=:black, linestyle=:solid,
          yscale=:log10, linewidth=2, linealpha=0.75,
          label="standard smoothed periodogram", legend=:topleft)

    x, y_ar = ar_periodogram(X, "hamming", w1)
    plot!(p[i], x, y_ar, linecolor=:blue, linestyle=:solid,
          yscale=:log10, linewidth=2, linealpha=0.75,
          label="AR smoothed periodogram", legend=:topleft)
end

Out[7]:
CHAPTER 55. ESTIMATION OF SPECTRA
Chapter 56

Additive Functionals

56.1 Contents

- Overview 56.2
- A Particular Additive Functional 56.3
- Dynamics 56.4
- Code 56.5

Co-authored with Chase Coleman and Balint Szoke

56.2 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 [47] 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 [46] (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 [46].
56.3 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 \textbf{first-order vector autoregression (VAR)}

\[
x_{t+1} = Ax_t + Bz_{t+1}
\]

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 (1)

In particular,

\[
y_{t+1} - y_t = \nu + Dx_t + Fz_{t+1}
\]

Here \( y_0 \sim \mathcal{N}(\mu_{y0}, \Sigma_{y0}) \) is a random initial condition.

The nonstationary random process \( \{y_t\}_{t=0}^{\infty} \) displays systematic but random \textit{arithmetic growth}.

56.3.1 A linear state space representation

One way to represent the overall dynamics is to use a \textbf{linear state space system}.

To do this, we set up state and observation vectors

\[
\hat{x}_t = \begin{bmatrix} 1 \\ x_t \\ y_t \end{bmatrix} \quad \text{and} \quad \hat{y}_t = \begin{bmatrix} x_t \\ y_t \end{bmatrix}
\]

Now we construct the state space system

\[
\begin{bmatrix} 1 \\ x_{t+1} \\ y_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & A & 0 \\ \nu & D' & 1 \end{bmatrix} \begin{bmatrix} 1 \\ x_t \\ y_t \end{bmatrix} + \begin{bmatrix} B \\ F' \end{bmatrix} z_{t+1}
\]

\[
\begin{bmatrix} x_t \\ y_t \end{bmatrix} = \begin{bmatrix} 0 & I & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ x_t \\ y_t \end{bmatrix}
\]
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.

## 56.4 Dynamics

Let's run some simulations to build intuition.

In doing so we’ll assume that \( z_{t+1} \) is scalar and that \( \tilde{x}_t \) follows a 4th-order scalar autoregression

\[
\tilde{x}_{t+1} = \phi_1\tilde{x}_t + \phi_2\tilde{x}_{t-1} + \phi_3\tilde{x}_{t-2} + \phi_4\tilde{x}_{t-3} + \sigma z_{t+1} \tag{3}
\]

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 (3) is not a first order system like (1), 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 (1) – (2) 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.

### 56.4.1 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 \([47]\).

All of these objects are computed using the code below.

### 56.4.2 Setup

In [1]:

```julia
using InstantiateFromURL
# optionally add arguments to force installation: instantiate = true,
# precompile = true

github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")
```
In [2]: using LinearAlgebra, Statistics

In [3]: using Distributions, Parameters, Plots, QuantEcon
   gr(fmt = :png);

In [4]: function AMF_LSS_VAR(A, B, D, F = nothing; ν = nothing)

   if B isa AbstractVector
      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 D isa Union{Adjoint, Transpose}
         D = convert(Matrix, D)
      end
   else
      nm = 1
      D = reshape(D, 1, length(D))
   end

   # set F
   if isnothing(F)
      F = zeros(nk, 1)
   elseif ndims(F) == 1
      F = reshape(F, length(F), 1)
   end

   # set ν
   if isnothing(ν)
      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
   lss = construct_ss(A, B, D, F, ν, nx, nk, nm)

   return (A = A, B = B, D = D, F = F, ν = ν, nx = nx, nk = nk, nm = nm, lss = lss)
end

AMF_LSS_VAR(A, B, D) =
   AMF_LSS_VAR(A, B, D, nothing, ν=nothing)
AMF_LSS_VAR(A, B, D, F, ν) =
   AMF_LSS_VAR(A, B, D, [F], ν=[ν])

function construct_ss(A, B, D, F,
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)
nym = I + zeros(nm, nm)
ny0m = zeros(nm, nm)
nyx0m = similar(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, nyx0m', nyx0m') # 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, I, nyx0m', nyx0m')   # selector for x_{t}
G2 = hcat(ny0c, ny0c, nyx0m, ny1m, ny0m) # selector for y_{t}
G3 = hcat(ny0c, ny0c, nyx0m, ny0m, ny1m) # selector for stationary
G4 = hcat(ny0c, ny0c, -g, ny0m, ny0m)     # selector for martingale
G5 = hcat(ny0c, ν, nyx0m, ny0m, ny0m)     # selector for trend
Gbar = vcat(G1, G2, G3, G4, G5)

# build LSS type
x0 = hcat(1, 0, nx0r, ny0r, ny0r)
S0 = zeros(length(x0), length(x0))
lss = LSS(Abar, Bbar, Gbar, zeros(nx+4nm, 1), x0, S0)

return lss
end

function additive_decomp(A, B, D, F, ν, nx)
    A_res = \( \begin{pmatrix} I & -A \\ I \end{pmatrix} \)
    g = D * A_res
    H = F .+ D * A_res * B

    return H, g
end

function multiplicative_decomp(A, B, D, F, ν, nx)
    H, g = additive_decomp(A, B, D, F, ν, nx)
    ν_tilde = ν .+ 0.5 * diag(H * H')
return H, g, ν_tilde
end

function loglikelihood_path(amf, x, y)
    @unpack A, B, D, F = amf
    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 = (logdet(FF) + k * log(2π)) * (1:T)
    return -(obssum + scalar) / 2
end

function loglikelihood(amf, x, y)
    llh = loglikelihood_path(amf, x, y)
    return llh[end]
end

function plot_additive(amf, T; npaths = 25, show_trend = true)
    # pull out right sizes so we know how to increment
    @unpack nx, nk, nm = amf
    # allocate space (nm is the number of additive functionals - we want npaths for each)
    mpath = zeros(nm*npaths, T)
    mbounds = zeros(2nm, T)
    spath = zeros(nm*npaths, T)
    sbounds = zeros(2nm, T)
    tpath = zeros(nm*npaths, T)
    ypath = zeros(nm*npaths, T)
    # simulate for as long as we wanted
    moment_generator = moment_sequence(amf.lss)
    # pull out population moments
    for (t, x) in enumerate(moment_generator)
        ymeans = x[2]
        yvar = x[4]
        # lower and upper bounds - for each additive functional
        for ii in 1:nm
            li, ui = 2(ii - 1) + 1, 2ii
            if sqrt(yvar[nx + nm + ii, nx + nm + ii]) != 0.0
                madd_dist = Normal(ymeans[nx + nm + ii], sqrt(yvar[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
                ...
```plaintext
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]
    sbounds[ui, t] = ymeans[nx + 2 nm + ii]
else
    error("standard error is negative")
end

end

t == T && break
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 = []
for ii in 0:nm - 1
    li, ui = npaths*(ii), npaths*(ii + 1)
    LI, UI = 2ii, 2(ii + 1)
    push!(add_figs,
        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))
end
return add_figs
end

function plot_multiplicative(amf, T, npaths = 25, show_trend = true)
    # pull out right sizes so we know how to increment
    @unpack nx, nk, nm = amf
    # matrices for the multiplicative decomposition
    H, g, ν̃ = multiplicative_decomp(A, B, D, F, ν, nx)

    # allocate space (nm is the number of functionals - we want npaths for each)
    mpath_mult = zeros(nm * npaths, T)
    mbounds_mult = zeros(2nm, T)
    spath_mult = zeros(nm * npaths, T)
```

sbounds_mult = zeros(2nm, T)
tpath_mult = zeros(nm * npaths, T)
ypath_mult = zeros(nm * npaths, T)

# simulate for as long as we wanted
moment_generator = moment_sequence(amf.lss)

# pull out population moments
for (t, x) in enumerate(moment_generator)
ymeans = x[2]
yvar = x[4]

# lower and upper bounds - for each multiplicative functional
for ii in 1:nm
    li, ui = 2(ii - 1) + 1, 2ii
    if yvar[nx + nm + ii, nx + nm + ii] != 0.0
        Mdist = LogNormal(ymean[nx + nm + ii] - 0.5t * 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] == 0.0
        mbounds_mult[li, t] = exp.(ymean[nx + nm + ii] - 0.5t *
                                diag(H * H')[ii])
        mbounds_mult[ui, t] = exp.(ymean[nx + nm + ii] - 0.5t *
                                     diag(H * H')[ii])
    else
        error("standard error is negative")
    end
    if yvar[nx + 2nm + ii, nx + 2nm + ii] != 0.0
        Sdist = LogNormal(-ymean[nx + 2nm + ii],
                           sqrt(yvar[nx + 2nm + ii, nx + 2nm + ii]))
        sbounds_mult[li, t] = quantile(Sdist, 0.01)
        sbounds_mult[ui, t] = quantile(Sdist, 0.99)
    elseif yvar[nx + 2nm + ii, nx + 2nm + ii] == 0.0
        sbounds_mult[li, t] = exp.(-ymean[nx + 2nm + ii])
        sbounds_mult[ui, t] = exp.(-ymean[nx + 2nm + ii])
    else
        error("standard error is negative")
    end
end
t = T && break

# 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, :])
        mpath_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+2nm + ii+1, :])
        tpath_mult[npaths * ii + n, :] =
            exp.(y[nx + 3nm + ii+1, :] + (1:T) * 0.5 * diag(H * H')[ii+1])
    end
end

mult_figs = []

for ii in 0:nm-1
    li, ui = npaths * ii, npaths * (ii + 1)
    LI, UI = 2ii, 2(ii + 1)
    push!(mult_figs,
        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:UI, :], sbounds_mult[LI:UI, :],
        horline = 1.0, show_trend=show_trend))
end

return mult_figs
end

function plot_martingales(amf, T, npaths = 25)

    # pull out right sizes so we know how to increment
    @unpack A, B, D, F, ν, nx, nk, nm = amf
    # matrices for the multiplicative decomposition
    H, g, ν_tilde = multiplicative_decomp(A, B, D, F, ν, nx)

    # allocate space (nm is the number of functionals - we want npaths for each)
    mpath_mult = zeros(nm * npaths, T)
    mbounds_mult = zeros(2nm, T)

    # simulate for as long as we wanted
    moment_generator = moment_sequence(amf.lss)

    # pull out population moments
    for (t, x) in enumerate(moment_generator)
        ymeans = x[2]
        yvar = x[4]

        # lower and upper bounds - for each functional
        for ii in 1:nn
            li, ui = 2(ii - 1) + 1, 2ii
            if yvar[nx + nm + ii, nx + nm + ii] != 0.0
                Mdist = LogNormal(ymeans[nx + nm + ii] - 0.5^2 * t * 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] == 0.0
                mbounds_mult[li, t] = ymeans[nx + nm + ii] - 0.5^2 * t * diag(H["H'"[ii])
            else
                error("standard error is negative")
            end
        end
        t == T && break
    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 = []
for ii in 0:nm-1
    li, ui = npaths*(ii), npaths*(ii + 1)
    LI, UI = 2ii, 2(ii + 1)
    push!(mart_figs,
        plot_martingale_paths(T, mpath_mult[li + 1:ui, :],
            mbounds_mult[LI + 1:UI, :]
        )
    end
end
return mart_figs
end

function plot_given_paths(T, ypath, mpath, spath, tpath, mbounds, sbounds; horline = 0.0, show_trend = true)
    # allocate space
    trange = 1:T
    # allocate transpose
    mpath' = Matrix(mpath')
    # create figure
    plots=plot(layout = (2, 2), size = (800, 800))
    # plot all paths together
    plot!(plots[1], trange, ypath[1, :], label = "y_t", color = :black)
    plot!(plots[1], trange, mpath[1, :], label = "m_t", color = :magenta)
    plot!(plots[1], trange, spath[1, :], label = "s_t", color = :green)
    if show_trend
        plot!(plots[1], trange, tpath[1, :], label = "t_t", color = :red)
    end
    plot!(plots[1], trange, ypath[1, :], label = ", linestyle=:dash,
        seriestype = :hline, [horline], color = :black,"n"
    )
    plot!(plots[1], title = "One Path of All Variables", legend=:topleft)
    # plot martingale component
    plot!(plots[2], trange, mpath[1, :], color = :magenta, label = ""
    )
    plot!(plots[2], trange, mpath', alpha = 0.45, color = :magenta, label = ""
    )
    ub = mbounds[2, :]
lb = mbounds[1, :]
plot!(plots[2], ub, fillrange = [lb, ub], alpha = 0.25, color = :magenta, label = ""

plot!(plots[2], seriestype = :hline, [horline], color = :black, linestyle = :dash, label = ""

plot!(plots[2], title = "Martingale Components for Many Paths")

# plot stationary component
plot!(plots[3], spath[1, :], color = :green, label = ""
plot!(plots[3], Matrix(spath'), alpha = 0.25, color = :green, label = ""

ub = sbounds[2, :]
lb = sbounds[1, :]
plot!(plots[3], ub, fillrange = [lb, ub], alpha = 0.25, color = :green, label = ""

plot!(plots[3], seriestype = :hline, [horline], color = :black, linestyle = :dash, label = ""

plot!(plots[3], title = "Stationary Components for Many Paths")

# plot trend component
if show_trend == true
    plot!(plots[4], Matrix(tpath'), color = :red, label = ""

end
plot!(plots[4], seriestype = :hline, [horline], color = :black, linestyle = :dash, label = ""

plot!(plots[4], title = "Trend Components for Many Paths")

return plots
end

function plot_martingale_paths(T, mpath, mbounds;
    horline = 1, show_trend = false)
    # allocate space
    trange = 1:T

    # create the plot
    plt = plot()

    # plot martingale component
    ub = mbounds[2, :]
lb = mbounds[1, :]
    plot!(plt, lb, fillrange = [lb, ub], alpha = 0.25, color = :magenta, label = ""

    plot!(plt, seriestype = :hline, [horline], color = :black, linestyle = :dash, label = ""

    plot!(plt, trange, Matrix(mpath'), linewdth=0.25, color = :black, label = ""

    return plt
end

Out[4]: plot_martingale_paths (generic function with 1 method)
For now, we just plot $y_t$ and $x_t$, postponing until later a description of exactly how we compute them.

\begin{verbatim}
In [5]: \phi_1, \phi_2, \phi_3, \phi_4 = 0.5, -0.2, 0, 0.5
\sigma = 0.01
\nu = 0.01 # growth rate

# A matrix should be n x n
A = [\phi_1 \phi_2 \phi_3 \phi_4; 1 0 0 0; 0 1 0 0; 0 0 1 0]

# B matrix should be n x k
B = [\sigma, 0, 0, 0]

D = [1 0 0 0] * A
F = [1, 0, 0, 0]  vec(B)

amf = AMF_LSS_VAR(A, B, D, F, \nu)

T = 150
x, y = simulate(amf.lss, T)

plots = plot(layout = (2, 1))

plot!(plots[1], 1:T, y[amf.nx + 1, :], color = :black, lw = 2, label = "")
plot!(plots[1], title = "A particular path of y_t")
plot!(plots[2], 1:T, y[1, :], color = :green, lw = 2, label = "")
plot!(plots[2], seriestype = :hline, [0], color = :black, lw = 2, linestyle = :dashdot, label = ")
plot!(plots[2], title = "Associated path of x_t")

plot(plots)
\end{verbatim}

Out[5]:
Notice the irregular but persistent growth in $y_t$.

### 56.4.3 Decomposition

Hansen and Sargent [46] 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 (1) and (2), we first construct the matrices

$$H := F + B'(I - A')^{-1}D$$
$$g := D'(I - A)^{-1}$$

Then the Hansen-Scheinkman [47] decomposition is

$$y_t = t\nu + \sum_{j=1}^{t} H z_j - g x_t + \frac{g x_0 + y_0}{t}$$

At this stage you should pause and verify that $y_{t+1} - y_t$ satisfies (2).

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 (1) and (2) and with the definitions just given,

\[
\begin{bmatrix}
1 \\
\tau_{t+1} \\
m_{t+1}
\end{bmatrix}
= 
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
\nu & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
1 \\
x_t \\
y_t \\
\tau_t \\
m_t
\end{bmatrix}
+ 
\begin{bmatrix}
0 \\
B \\
F' \\
H'
\end{bmatrix}
\begin{bmatrix}
1 \\
z_{t+1}
\end{bmatrix}
\]

and

\[
\begin{bmatrix}
x_t \\
y_t \\
\tau_t \\
m_t
\end{bmatrix}
= 
\begin{bmatrix}
0 & 0 & I & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & \nu & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0
\end{bmatrix}
\begin{bmatrix}
1 \\
x_t \\
y_t \\
\tau_t \\
m_t
\end{bmatrix}
\]

With

\[
\tilde{x} := 
\begin{bmatrix}
1 \\
x_t \\
y_t \\
m_t
\end{bmatrix}
\quad \text{and} \quad
\tilde{y} := 
\begin{bmatrix}
x_t \\
y_t \\
\tau_t \\
m_t
\end{bmatrix}
\]

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.

### 56.5 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 you will generate (modulo randomness) the plot.
When we plot multiple realizations of a component in the 2nd, 3rd, and 4th panels, we also plot population 95% probability coverage sets computed using the LSS 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

### 56.5.1 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.
CHAPTER 56. ADDITIVE FUNCTIONALS

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

\[
\frac{M_t}{M_0} = \exp(\tilde{\nu}t) \left( \frac{\tilde{M}_t}{\tilde{M}_0} \right) \left( \frac{\tilde{e}(X_0)}{\tilde{e}(x_t)} \right)
\]

where

\[
\tilde{\nu} = \nu + \frac{H \cdot H}{2}, \quad \tilde{M}_t = \exp \left( \sum_{j=1}^{t} \left( H \cdot z_j - \frac{H \cdot H}{2} \right) \right), \quad \tilde{M}_0 = 1
\]

and

\[
\tilde{e}(x) = \exp[g(x)] = \exp[D'(I - A)^{-1}x]
\]

An instance of 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

\[
\text{In [7]: } \text{plt = plot_multiplicative(afm, T)}
\]

\[
\text{plt[1]}
\]

\[
\text{Out[7]:}
\]
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.

56.5.2 A peculiar large sample property

Hansen and Sargent [46] (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:

\begin{verbatim}
In [8]: plt = plot_martingales(amf, 12000)
plt[1]
\end{verbatim}

Out[8]:
Martingale components for many paths of $y_{(ii + 1)}$
Chapter 57

Multiplicative Functionals

57.1 Contents

- Overview 57.2
- A Log-Likelihood Process 57.3
- Benefits from Reduced Aggregate Fluctuations 57.4

Co-authored with Chase Coleman and Balint Szoke

57.2 Overview

This lecture is a sequel to the lecture on additive functionals.

That lecture

1. defined a special class of additive functionals driven by a first-order vector VAR
2. by taking the exponential of that additive functional, created an associated multiplicative functional

This lecture uses this special class to create and analyze two examples

- A log likelihood process, an object at the foundation of both frequentist and Bayesian approaches to statistical inference.
- A version of Robert E. Lucas’s [70] and Thomas Tallarini’s [103] approaches to measuring the benefits of moderating aggregate fluctuations.

57.3 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

\[
\log L_t(\theta) = -\frac{1}{2} \sum_{j=1}^{t} (y_j - y_{j-1} - Dx_{j-1})'(FF')^{-1}(y_j - y_{j-1} - Dx_{j-1}) \\
-\frac{t}{2} \log \det(FF') - \frac{kt}{2} \log(2\pi)
\]

Let’s consider the case of a scalar process in which \( A, B, D, F \) are scalars and \( z_{t+1} \) is a scalar stochastic process.

We let \( \theta_o \) denote the “true” values of \( \theta \), meaning the values that generate the data.

For the purposes of this exercise, set \( \theta_o = (A, B, D, F) = (0.8, 1, 0.5, 0.2) \).

Set \( x_0 = y_0 = 0 \).

### 57.3.1 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 struct to work.

### 57.3.2 Setup

In [1]: using InstantiateFromURL
   # optionally add arguments to force installation: instantiate = true, precompile = true
   github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

In [2]: using LinearAlgebra, Statistics
   using Distributions, Parameters, Plots, QuantEcon
   import Distributions: loglikelihood
   gr(fmt = :png);

In [3]: AMF_LSS_VAR = @with_kw (A, B, D, F = @.0, ν = @.0, lss = construct_ss(A, B, D, F, ν))

   function construct_ss(A, B, D, F, ν)
       H, g = additive_decomp(A, B, D, F)
   end

   function loglikelihood(θ, θ₀, lss)
       "loglikelihood of \{y_s\}_{s=1}^t as specified in (5.2.2)
       with respect to \( \theta_o \) = (A, B, D, F) = (0.8, 1, 0.5, 0.2)."
   end
# Build A matrix for LSS
# Order of states is: [1, t, xt, yt, mt]
A1 = [1 0 0 0 0]  # Transition for 1
A2 = [1 1 0 0 0]  # Transition for t
A3 = [0 0 A 0 0]  # Transition for x_{t+1}
A4 = [ν 0 D 1 0]  # Transition for y_{t+1}
A5 = [0 0 0 0 1]  # 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]  # Selector for x_{t}
G2 = [0 0 0 1 0]  # Selector for y_{t}
G3 = [0 0 0 0 1]  # Selector for martingale
G4 = [0 0 -g 0 0]  # Selector for stationary
G5 = [0 ν 0 0 0]  # Selector for trend
Gbar = vcat(G1, G2, G3, G4, G5)

# Build LSS struct
x0 = [0, 0, 0, 0, 0]
S0 = zeros(5, 5)
return LSS(Abar, Bbar, Gbar, mu_0 = x0, Sigma_0 = S0)
end

function additive_decomp(A, B, D, F)
    A_res = 1 / (1 - A)
    g = D * A_res
    H = F + D * A_res * B

    return H, g
end

function multiplicative_decomp(A, B, D, F, ν)
    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, x, y)
    @unpack A, B, D, F = amf
    T = length(y)
    FF = F^2
    FFinv = inv(FF)
    obs = temp .* FFinv .* temp
    obssum = cumsum(obs)
    scalar = (log(FF) + log(2*pi)) * (1:T-1)
    return -0.5 * (obssum + scalar)
end

function loglikelihood(amf, x, y)
    llh = loglikelihood_path(amf, x, y)
    return llh[end]
end
The heavy lifting is done inside the AMF_LSS_VAR struct. The following code adds some simple functions that make it straightforward to generate sample paths from an instance of AMF_LSS_VAR.

```python
function simulate_xy(amf, T)
    foo, bar = simulate(amf.lss, T)
    x = bar[1, :]
    y = bar[2, :]
    return x, y
end

function simulate_paths(amf, T = 150, I = 5000)
    # Allocate space
    storeX = zeros(I, T)
    storeY = zeros(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, T = 150)
    # Allocate Space
    xmean = zeros(T)
    ymean = zeros(T)

    # Pull out moment generator
    moment_generator = moment_sequence(amf.lss)
    for (tt, x) = enumerate(moment_generator)
        ymeans = x[2]
        xmean[tt] = ymeans[1]
        ymean[tt] = ymeans[2]
        if tt == T
            break
        end
    end
    return xmean, ymean
end
```

Now that we have these functions in our toolkit, 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 $\{x^i_t, y^i_t\}_{t=0}^T$ for $i = 1, ..., I$.

Then we compute averages of $\frac{1}{T} \sum_{t=0}^{T} x^i_t$ and $\frac{1}{T} \sum_{t=0}^{T} y^i_t$ across the sample paths and compare them with the population means of $x_t$ and $y_t$. 

Out[3]: loglikelihood (generic function with 4 methods)

Out[4]: population_means (generic function with 2 methods)
57.3. A LOG-LIKELIHOOD PROCESS

Here goes

In [5]:

```python
F = 0.2
amf = AMF_LSS_VAR(A = 0.8, B = 1.0, D = 0.5, F = F)

T = 150
I = 5000

# Simulate and compute sample means
Xit, Yit = simulate_paths(amf, T, I)
Xmean_t = mean(Xit, dims = 1)
Ymean_t = mean(Yit, dims = 1)

# Compute population means
Xmean_pop, Ymean_pop = population_means(amf, T)

# Plot sample means vs population means
plt_1 = plot(Xmean_t', color = :blue, label = "1/I sum_i x_t^i")
plot!(plt_1, Xmean_pop, color = :black, label = "E x_t")
plot!(plt_1, title = "x_t", xlim = (0, T), legend = :bottomleft)

plt_2 = plot(Ymean_t', color = :blue, label = "1/I sum_i x_t^i")
plot!(plt_2, Ymean_pop, color = :black, label = "E y_t")
plot!(plt_2, title = "y_t", xlim = (0, T), legend = :bottomleft)
plot(plt_1, plt_2, layout = (2, 1), size = (800,500))
```

Out[5]:

![Graphs showing x_t and y_t with simulated and population means](image)

57.3.3 Simulating log-likelihoods

Our next aim is to write a program to simulate \( \{ \log L_i \mid \theta_o \}_{i=1}^T \).
We want as inputs to this program the same sample paths \( \{ x_i^t, y_i^t \}_{t=0}^T \) that we have already computed.

We now want to simulate \( I = 5000 \) paths of \( \{ \log L_i^t \mid \theta_0 \}_{t=1}^T \).

- For each path, we compute \( \log L_i^T / T \).
- We also compute \( \frac{1}{T} \sum_{t=1}^T \log L_i^t / T \).

Then we to compare these objects.

Below we plot the histogram of \( \log L_i^T / T \) for realizations \( i = 1, \ldots, 5000 \).

```python
In [6]: function simulate_likelihood(amf, Xit, Yit)
# Get size
I, T = size(Xit)

# Allocate space
LLit = zeros(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}
LLit = simulate_likelihood(amf, Xit, Yit)

LLT = 1 / T * LLit[:, end]
LLmean_t = mean(LLT)

plot(seriestyle = :histogram, LLT, label = "")
plot!(title = "Distribution of (I/T)log(L_T|theta_0")
vline!(LLmean_t, linestyle = :dash, color = :black, lw = 2, alpha = 0.6, label = "")
```

Out[6]:
Notice that the log likelihood is almost always nonnegative, implying that \( L_t \) is typically bigger than 1.

Recall that the likelihood function is a pdf (probability density function) and not a probability measure, so it can take values larger than 1.

In the current case, the conditional variance of \( \Delta y_{t+1} \), which equals \( FF^T = 0.04 \), is so small that the maximum value of the pdf is 2 (see the figure below).

This implies that approximately 75% of the time (a bit more than one sigma deviation), we should expect the increment of the log likelihood to be nonnegative.

Let’s see this in a simulation

```python
In [7]: normdist = Normal(0, F)
    mult = 1.175
    println("The pdf at +/- $mult sigma takes the value:")
    println(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:")
    println(frac_nonnegative)
```

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: 0.7601202702702703

Let’s also plot the conditional pdf of \( \Delta y_{t+1} \)
In [8]: xgrid = range(-1, 1, length = 100)
println("The pdf at +/- one sigma takes the value: $(pdf(normdist, F)) ")
plot(xgrid, pdf.(normdist, xgrid), label = "")
plot!(title = "Conditional pdf $f(Delta y_{(t+1)} \mid x_t)"")

The pdf at +/- one sigma takes the value: 1.2098536225957168

Out[8]:

57.3.4 An alternative parameter vector

Now consider alternative parameter vector $\theta_1 = [A, B, D, F] = [0.9, 1.0, 0.55, 0.25]$.

We want to compute $\{\log L_t \mid \theta_1\}_{t=1}^T$.

The $x_t, y_t$ inputs to this program should be exactly the same sample paths $\{x_t^i, y_t^i\}_{t=0}^T$ that we computed above.

This is because we want to generate data under the $\theta_o$ probability model but evaluate the likelihood under the $\theta_1$ model.

So our task is to use our program to simulate $I = 5000$ paths of $\{\log L_t^i \mid \theta_1\}_{t=1}^T$.

- For each path, compute $\frac{1}{T} \log L_T^i$.
- Then compute $\frac{1}{T} \sum_{t=1}^T \frac{1}{T} \log L_t^i$.

We want to compare these objects with each other and with the analogous objects that we computed above.

Then we want to interpret outcomes.

A function that we constructed can handle these tasks.
The only innovation is that we must create an alternative model to feed in.

We will creatively call the new model `amf2`.

We make three graphs

- the first sets the stage by repeating an earlier graph
- the second contains two histograms of values of log likelihoods of the two models over the period $T$
- the third compares likelihoods under the true and alternative models

Here’s the code

```
In [9]: # Create the second (wrong) alternative model
amf2 = AMF_LSS_VAR(A = 0.9, B = 1.0, D = 0.55, F = 0.25) # parameters for $\theta_1$ closer to $\theta_0$
# 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)

plot(seriestyle = :histogram, LLT2, label = "")
vline!(LLmean_t2, color = :black, lw = 2, linestyle = :dash, alpha = 0.6, label = "")
plot!(title = "Distribution of $(1/T)log(L_T | \theta_1)"")
```

```
Out[9]:
```

Let’s see a histogram of the log-likelihoods under the true and the alternative model (same sample paths)
In [10]: plot(seriestyle = :histogram, LLT, bin = 50, alpha = 0.5, label = "True",
     normed = true)
    plot!(seriestyle = :histogram, LLT2, bin = 50, alpha = 0.5, label = "Alternative",
     normed = true)
    vline!([mean(LLT)], color = :black, lw = 2, linestyle = :dash, label = "")
    vline!([mean(LLT2)], color = :black, lw = 2, linestyle = :dash, label = ")

Out[10]:

Now we'll plot the histogram of the difference in log likelihood ratio

In [11]: LLT_diff = LLT - LLT2
    plot(seriestyle = :histogram, LLT_diff, bin = 50, label = "")
    plot!(title = "(1/T)[log(L|_i | theta_0) - log(L|_i |theta_1)]")

Out[11]:
57.3.5 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, we’ll use some of the code prepared in this lecture to illustrate mistake probabilities)

57.4 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., [7], [42], [41])
- benefits of eliminating aggregate fluctuations in representative agent macro models (e.g., [103], [70])

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 \( N(0, I) \) 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

\[ U = \exp(-\delta) [I - \exp(-\delta)A']^{-1} D \]

and

\[ u = \exp\left(-\frac{\delta}{1 - \exp(-\delta)}\right) \nu + \frac{(1-\gamma)}{2} \exp\left(-\frac{\delta}{1 - \exp(-\delta)}\right) \left[ D' [I - \exp(-\delta)A]^{-1} B + F \right]^2, \]

Here \( \gamma \geq 1 \) is a risk-aversion coefficient and \( \delta > 0 \) is a rate of time preference.

### 57.4.1 Consumption as a multiplicative process

We begin by showing that consumption is a multiplicative functional with representation

\[ \frac{c_t}{c_0} = \exp(\tilde{\nu}t) \begin{pmatrix} \tilde{M}_t \\ \tilde{M}_0 \end{pmatrix} \begin{pmatrix} \tilde{e}(x_0) \\ \tilde{e}(x_t) \end{pmatrix} \]  

where \( \begin{pmatrix} \tilde{M}_t \\ \tilde{M}_0 \end{pmatrix} \) is a likelihood ratio process and \( \tilde{M}_0 = 1 \).

At this point, as an exercise, we ask the reader please to verify the follow formulas for \( \tilde{\nu} \) and \( \tilde{e}(x_t) \) as functions of \( A, B, D, F \):

\[ \tilde{\nu} = \nu + \frac{H \cdot H}{2} \]

and

\[ \tilde{e}(x) = \exp[g(x)] = \exp[D'(I - A)^{-1}x] \]

### 57.4.2 Simulating a likelihood ratio process again

Next, we want a program to simulate the likelihood ratio process \( \{\tilde{M}_t\}_{t=0}^{\infty} \).
In particular, we want to simulate 5000 sample paths of length \( T = 1000 \) for the case in which \( x \) is a scalar and \( [A, B, D, F] = [0.8, 0.001, 1.0, 0.01] \) and \( \nu = 0.005 \).

After accomplishing this, we want to display a histogram of \( \tilde{M}_T^i \) for \( T = 1000 \).

Here is code that accomplishes these tasks

```plaintext
In [12]: function simulate_martingale_components(amf, T = 1_000, I = 5_000)
    # Get the multiplicative decomposition
    @unpack A, B, D, F, ν, lss = amf
    ν, H, g = multiplicative_decomp(A, B, D, F, ν)
    # Allocate space
    add_mart_comp = zeros(I, T)
    # Simulate and pull out additive martingale component
    for i in 1:I
        foo, bar = simulate(lss, T)
        # Martingale component is third component
        add_mart_comp[i, :] = bar[3, :]
    end
    mul_mart_comp = exp.(add_mart_comp' .- (0:T-1) * H^2 / 2)

    return add_mart_comp, mul_mart_comp
end

# Build model
amf_2 = AMF_LSS_VAR(A = 0.8, B = 0.001, D = 1.0, F = 0.01, ν = 0.005)
amc, mmc = simulate_martingale_components(amf_2, 1_000, 5_000)
amcT = amc[:, end]
mmcT = mmc[:, end]

println("The (min, mean, max) of additive Martingale component in period T is")
println("\t\t($\text{minimum}(amcT)), ($\text{mean}(amcT)), ($\text{maximum}(amcT))")
println("The (min, mean, max) of multiplicative Martingale component in period T is")
println("\t\t($\text{minimum}(mmcT)), ($\text{mean}(mmcT)), ($\text{maximum}(mmcT))")

The (min, mean, max) of additive Martingale component in period T is (-1.5976378832036093, 0.011688395462056569, 1.8548306466611928)
The (min, mean, max) of multiplicative Martingale component in period T is (0.18086120172214815, 1.0115798917139234, 5.711279885712145)

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:

```
In [13]: plot(seriestype = :histogram, amcT, bin = 25, normed = true, label = "")
plot!(title = "Histogram of Additive Martingale Component")
```

Out[13]:

```
Here’s a histogram of the multiplicative martingale component:
```

```
In [14]: plot(seriestype = :histogram, mmcT, bin = 25, normed = true, label = "")
plot!(title = "Histogram of Multiplicative Martingale Component")
```

Out[14]:

```
Here’s a histogram of the multiplicative martingale component:
```
57.4.3 Representing the likelihood ratio process

The likelihood ratio process \( \{\tilde{M}_t\}_{t=0}^{\infty} \) can be represented as

\[
\tilde{M}_t = \exp\left( \sum_{j=1}^{t} \left( H \cdot z_j - \frac{H \cdot H}{2} \right) \right), \quad \tilde{M}_0 = 1,
\]

where 

\[
H = [F + B'(I - A')^{-1}D].
\]

It follows that \( \log \tilde{M}_t \sim \mathcal{N}\left( -\frac{tH \cdot H}{2}, tH \cdot H \right) \) and that consequently \( \tilde{M}_t \) is log normal.

Let’s plot the probability density functions for \( \log \tilde{M}_t \) for \( t = 100, 500, 1000, 10000, 100000 \).

Then let’s use the plots to investigate how these densities evolve through time.

We will plot the densities of \( \log \tilde{M}_t \) for different values of \( t \).

Here is some code that tackles these tasks

In [15]: function Mtilde_t_density(amf, t; xmin = 1e-8, xmax = 5.0, npts = 5000)

    # Pull out the multiplicative decomposition
    \( \tilde{v}_t, H, g = \)
    \( H2 = H^*H \)

    # The distribution
    \( \text{mdist} = \text{LogNormal}(\cdot t * H2 / 2, \sqrt{t * H2}) \)
    \( x = \text{range}(xmin, xmax, length = npts) \)
    \( p = \text{pdf}(\text{mdist}, x) \)

    \( \text{return} x, p \)
function logMtilde_t_density(amf, t; xmin = -15.0, xmax = 15.0, npts = 5000)

    # Pull out the multiplicative decomposition
    @unpack A, B, D, F, ν = amf
    \( \nu \tilde{t} \), H, g = multiplicative_decomp(A, B, D, F, ν)
    \( H^2 = H \times H \)

    # The distribution
    lmdist = Normal(-t * H2 / 2, sqrt(t * H2))
    x = range(xmin, xmax, length = 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]

    # plot_title = "Densities of M_t^tilda" is required, however, plot_title is not yet supported in Plots
plots = plot(layout = (3,2), size = (600,800))
for (it, dens_t) in enumerate(dens_to_plot)
    x, pdf = dens_t
    plot!(plots[it], title = "Density for time \( (time_to_plot[\text{it}]) \)"
for (it, dens_t) in enumerate(ddens_to_plot)
    x, pdf = dens_t
    plot!(plots[it], pdf, fillrange = [[0], pdf], label = "")
end
plot(plots)
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 [103], [70]) 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 (1) would be willing to sacrifice to enjoy the consumption stream

$$\frac{c_L}{c_0} = \exp(\tilde{\nu}t)$$

rather than the stream described by equation (2).

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

```python
In [16]: function Uu(amf, \delta, \gamma)
    @unpack A, B, D, F, \nu = amf
    \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(-\tilde{\nu}) * resolv * D
    u_risky = \exp(-\tilde{\nu}) / (1 - \exp(-\tilde{\nu})) * (\nu + 0.5 * (1 - \gamma) * (vect^2))
    U_det = 0
    u_det = \exp(-\tilde{\nu}) / (1 - \exp(-\tilde{\nu})) * \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)
```

Out[16]: (4.54129843114712, 0.24220854072375247, 0, 0.25307727077652764)

The values of the two processes are

$$\log V^r_0 = \log c^r_0 + U^r x_0 + u^r$$

$$\log V^d_0 = \log c^d_0 + U^d x_0 + u^d$$

We look for the ratio $\frac{c^r_0 - c^d_0}{c^r_0}$ that makes $\log V^r_0 - \log V^d_0 = 0$
\[
\log V_r^r - \log V_d^d + \log c_0^r - \log c_0^d = (U_r - U_d)x_0 + u_r - u_d = 0
\]

Hence, the implied percentage reduction in \(c_0\) that the representative consumer would accept is given by

\[
\frac{c_0^d}{c_0^r} = \exp((U_r - U_d)x_0 + u_r - u_d)
\]

Let’s compute this

\[
\text{perc_reduct} = 100 \times (1 - \exp(\log VC_r - \log VC_d))
\]

\[
\text{perc_reduct} = 1.0809878812017448
\]

We find that the consumer would be willing to take a percentage reduction of initial consumption equal to around 1.081.
Chapter 58

Classical Control with Linear Algebra

58.1 Contents

- Overview 58.2
- A Control Problem 58.3
- Finite Horizon Theory 58.4
- The Infinite Horizon Limit 58.5
- Undiscounted Problems 58.6
- Implementation 58.7
- Exercises 58.8

58.2 Overview

In an earlier lecture Linear Quadratic Dynamic Programming Problems we have studied how to solve a special class of dynamic optimization and prediction problems by applying the method of dynamic programming. In this class of problems

- the objective function is quadratic in states and controls
- the one-step transition function is linear
- shocks are i.i.d. Gaussian or martingale differences

In this lecture and a companion lecture Classical Filtering with Linear Algebra, we study the classical theory of linear-quadratic (LQ) optimal control problems.

The classical approach does not use the two closely related methods – dynamic programming and Kalman filtering – that we describe in other lectures, namely, Linear Quadratic Dynamic Programming Problems and A First Look at the Kalman Filter.

Instead they use either.

- z-transform and lag operator methods, or.
- matrix decompositions applied to linear systems of first-order conditions for optimum problems.
In this lecture and the sequel Classical Filtering with Linear Algebra, we mostly rely on elementary linear algebra.

The main tool from linear algebra we’ll put to work here is LU decomposition.

We’ll begin with discrete horizon problems.

Then we’ll view infinite horizon problems as appropriate limits of these finite horizon problems.

Later, we will examine the close connection between LQ control and least squares prediction and filtering problems.

These classes of problems are connected in the sense that to solve each, essentially the same mathematics is used.

58.2.1 References

Useful references include [107], [44], [82], [6], and [80].

58.2.2 Setup

In [1]: using InstantiateFromURL
   # optionally add arguments to force installation: instantiate = true,[]
   # precompile = true
   github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

In [2]: using Polynomials, Plots, Random, Parameters
   using LinearAlgebra, Statistics

58.3 A Control Problem

Let $L$ be the lag operator, so that, for sequence $\{x_t\}$ we have $Lx_t = x_{t-1}$.

More generally, let $L^k x_t = x_{t-k}$ with $L^0 x_t = x_t$ and

$$d(L) = d_0 + d_1 L + \ldots + d_m L^m$$

where $d_0, d_1, \ldots, d_m$ is a given scalar sequence.

Consider the discrete time control problem

$$\max_{\{y_t\}} \lim_{N \to \infty} \sum_{t=0}^{N} \beta^t \left\{ a_t y_t - \frac{1}{2} h y_t^2 - \frac{1}{2} [d(L)y_t]^2 \right\},$$

where

- $h$ is a positive parameter and $\beta \in (0, 1)$ is a discount factor
- $\{a_t\}_{t \geq 0}$ is a sequence of exponential order less than $\beta^{-1/2}$, by which we mean
  $$\lim_{t \to \infty} \beta^{t/2} a_t = 0$$

Maximization in (1) is subject to initial conditions for $y_{-1}, y_{-2}, \ldots, y_{-m}$.

Maximization is over infinite sequences $\{y_t\}_{t \geq 0}$.
58.3.1 Example

The formulation of the LQ problem given above is broad enough to encompass many useful models.

As a simple illustration, recall that in lqcontrol we consider a monopolist facing stochastic demand shocks and adjustment costs.

Let’s consider a deterministic version of this problem, where the monopolist maximizes the discounted sum

$$\sum_{t=0}^{\infty} \beta^t \pi_t$$

and

$$\pi_t = p_t q_t - c q_t - \gamma (q_{t+1} - q_t)^2$$

with $$p_t = \alpha_0 - \alpha_1 q_t + d_t$$

In this expression, $$q_t$$ is output, $$c$$ is average cost of production, and $$d_t$$ is a demand shock.

The term $$\gamma (q_{t+1} - q_t)^2$$ represents adjustment costs.

You will be able to confirm that the objective function can be rewritten as (1) when

- $$a_t := \alpha_0 + d_t - c$$
- $$h := 2\alpha_1$$
- $$d(L) := \sqrt{2\gamma} (I - L)$$

Further examples of this problem for factor demand, economic growth, and government policy problems are given in ch. IX of [94].

58.4 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 (1) are not sufficient for maximizing (1))

We begin by

1. fixing $$N > m$$,
2. differentiating the finite version of (1) 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
\[ J = \sum_{t=0}^{N} \beta^t [d(L)y_t][d(L)y_t] \]
\[ = \sum_{t=0}^{N} \beta^t (d_0 y_t + d_1 y_{t-1} + \cdots + d_m y_{t-m}) (d_0 y_t + d_1 y_{t-1} + \cdots + d_m y_{t-m}) \]

Differentiating \( J \) with respect to \( y_t \) for \( t = 0, 1, \ldots, N-m \) gives

\[
\frac{\partial J}{\partial y_t} = 2\beta^t d_0 d(L)y_t + 2\beta^{t+1} d_1 d(L)y_{t+1} + \cdots + 2\beta^{t+m} d_m d(L)y_{t+m}
\]
\[ = 2\beta^t (d_0 + d_1 \beta L^{-1} + d_2 \beta^2 L^{-2} + \cdots + d_m \beta^m L^{-m}) d(L)y_t \]

We can write this more succinctly as

\[
\frac{\partial J}{\partial y_t} = 2\beta^t d(\beta L^{-1}) d(L)y_t \quad (2)
\]

Differentiating \( J \) with respect to \( y_t \) for \( t = N-m + 1, \ldots, N \) gives

\[
\frac{\partial J}{\partial y_t} = 2\beta^N d_0 d(L)y_N
\]
\[
\frac{\partial J}{\partial y_{N-1}} = 2\beta^{N-1} [d_0 + \beta d_1 L^{-1}] d(L)y_{N-1}
\]
\[ \vdots \]
\[
\frac{\partial J}{\partial y_{N-m+1}} = 2\beta^{N-m+1} [d_0 + \beta L^{-1} d_1 + \cdots + \beta^{m-1} L^{-m+1} d_m] d(L)y_{N-m+1} \quad (3)
\]

With these preliminaries under our belts, we are ready to differentiate (1).

Differentiating (1) with respect to \( y_t \) for \( t = 0, \ldots, N-m \) gives the Euler equations

\[ [h + d(\beta L^{-1}) d(L)]y_t = a_t, \quad t = 0, 1, \ldots, N-m \quad (4) \]

The system of equations (4) form a \( 2 \times m \) order linear difference equation that must hold for the values of \( t \) indicated.

Differentiating (1) with respect to \( y_t \) for \( t = N-m + 1, \ldots, N \) gives the terminal conditions

\[
\beta^N (a_N - h y_N - d_0 d(L)y_N) = 0
\]
\[
\beta^{N-1} \left( a_{N-1} - h y_{N-1} - \left( d_0 + \beta d_1 L^{-1} \right) d(L)y_{N-1} \right) = 0
\]
\[ \vdots \]
\[
\beta^{N-m+1} \left( a_{N-m+1} - h y_{N-m+1} - \left( d_0 + \beta L^{-1} d_1 + \cdots + \beta^{m-1} L^{-m+1} d_m \right) d(L)y_{N-m+1} \right) = 0 \quad (5)
\]

In the finite \( N \) problem, we want simultaneously to solve (4) subject to the \( m \) initial conditions \( y_{-1}, \ldots, y_{-m} \) and the \( m \) terminal conditions (5).

These conditions uniquely pin down the solution of the finite \( N \) problem.
That is, for the finite $N$ problem, conditions (4) and (5) are necessary and sufficient for a maximum, by concavity of the objective function.

Next we describe how to obtain the solution using matrix methods.

58.4.1 Matrix Methods

Let’s look at how linear algebra can be used to tackle and shed light on the finite horizon LQ control problem.

A Single Lag Term

Let’s begin with the special case in which $m = 1$.

We want to solve the system of $N + 1$ linear equations

\begin{align}
    [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
\end{align}

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 $y_{N-1}$.

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 (6) as the matrix equation

\begin{align}
    \begin{bmatrix}
        (\phi_0 - d_1^2) & \phi_1 & 0 & 0 & \ldots & \ldots & 0 \\
        \beta \phi_1 & \phi_0 & \phi_1 & 0 & \ldots & \ldots & 0 \\
        0 & \beta \phi_1 & \phi_0 & \phi_1 & \ldots & \ldots & 0 \\
        \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
        0 & \ldots & \ldots & \ldots & \ldots & \beta \phi_1 & \phi_0 \\
        0 & \ldots & \ldots & \ldots & 0 & \beta \phi_1 & \phi_0 \\
    \end{bmatrix}
    \begin{bmatrix}
        y_N \\
        y_{N-1} \\
        y_{N-2} \\
        \vdots \\
        y_1 \\
        y_0 \\
    \end{bmatrix}
    &=
    \begin{bmatrix}
        a_N \\
        a_{N-1} \\
        a_{N-2} \\
        \vdots \\
        a_1 \\
        a_0 - \phi_1 y_{N-1} \\
    \end{bmatrix}
\end{align}

or

\[ W \tilde{y} = \tilde{a} \]

Notice how we have chosen to arrange the $y_t$’s in reverse time order.

The matrix $W$ on the left side of (7) is “almost” a Toeplitz matrix (where each descending diagonal is constant).

There are two sources of deviation from the form of a Toeplitz matrix.

1. The first element differs from the remaining diagonal elements, reflecting the terminal condition.

2. The subdiagonal elements equal $\beta$ time the superdiagonal elements.
The solution of (8) can be expressed in the form

$$\bar{y} = W^{-1}\bar{a}$$  \hspace{1cm} (9)

which represents each element $y_t$ of $\bar{y}$ as a function of the entire vector $\bar{a}$.

That is, $y_t$ is a function of past, present, and future values of $a$’s, as well as of the initial condition $y_{-1}$.

**An Alternative Representation**

An alternative way to express the solution to (7) or (8) 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 (9), we obtain

$$U\bar{y} = L^{-1}\bar{a}$$  \hspace{1cm} (10)

Since $L^{-1}$ is lower triangular, this representation expresses $y_t$ as a function of

- lagged $y$’s (via the term $U\bar{y}$), and
- current and future $a$’s (via the term $L^{-1}\bar{a}$)

Because there are zeros everywhere in the matrix on the left of (7) 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, (10) has the form

$$\begin{bmatrix}
1 & U_{12} & 0 & 0 & \ldots & 0 & 0 \\
0 & 1 & U_{23} & 0 & \ldots & 0 & 0 \\
0 & 0 & 1 & U_{34} & \ldots & 0 & 0 \\
0 & 0 & 0 & 1 & \ldots & 0 & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & 0 & \ldots & 1 & U_{N,N+1} \\
0 & 0 & 0 & 0 & \ldots & 0 & 1 
\end{bmatrix} \begin{bmatrix}
y_N \\
y_{N-1} \\
y_{N-2} \\
y_{N-3} \\
\vdots \\
y_1 \\
y_0 
\end{bmatrix} = \begin{bmatrix}
y_N \\
y_{N-1} \\
y_{N-2} \\
y_{N-3} \\
\vdots \\
y_{N-N} \\
y_0 
\end{bmatrix}$$
where $L_{ij}$ is the $(i, j)$ element of $L^{-1}$ and $U_{ij}$ is the $(i, j)$ element of $U$.

Note how the left side for a given $t$ involves $y_t$ and one lagged value $y_{t-1}$ while the right side involves all future values of the forcing process $a_t, a_{t+1}, \ldots, a_N$.

### 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:

$$D_{jk} = d_0d_{k-j} + d_1d_{k-j+1} + \cdots + d_{j-1}d_{k-1}, \quad k \geq j$$

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 (4) and (5) can be expressed as:

$$(D_{m+1} + hI_{m+1}) \begin{bmatrix} y_N \\ y_{N-1} \\ \vdots \\ y_{N-m} \end{bmatrix} = \begin{bmatrix} a_N \\ a_{N-1} \\ \vdots \\ a_{N-m} \end{bmatrix} + M \begin{bmatrix} y_{N-m+1} \\ y_{N-m-2} \\ \vdots \\ y_{N-2m} \end{bmatrix}$$

where $M$ is $(m+1) \times m$ and

$$M_{ij} = \begin{cases} D_{i-j, m+1} & \text{for } i > j \\ 0 & \text{for } i \leq j \end{cases}$$

$$\begin{align*}
\phi_my_{N-1} + \phi_{m-1}y_{N-2} + \cdots + \phi_0y_{N-m-1} + \phi_1y_{N-m-2} + \\
\cdots + \phi_my_{N-2m-1} &= a_{N-m-1} \\
\phi_my_{N-2} + \phi_{m-1}y_{N-3} + \cdots + \phi_0y_{N-m-2} + \phi_1y_{N-m-3} + \\
\cdots + \phi_my_{N-2m-2} &= a_{N-m-2} \\
\vdots \\
\phi_my_{m+1} + \phi_{m-1}y_m + \cdots + \phi_0y_1 + \phi_1y_0 + \phi_my_{m-1} &= a_1 \\
\phi_my_{m} + \phi_{m-1}y_{m-1} + \phi_{m-2} + \cdots + \phi_0y_0 + \phi_1y_{1} + \cdots + \phi_my_{-m} &= a_0
\end{align*}$$

As before, we can express this equation as $Wy = \tilde{a}$.

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

\[
U \hat{y} = L^{-1} \hat{a}
\]

where \( L^{-1} \) is the element in the \((t, s)\) position of \( L \), and similarly for \( U \).

The left side of equation (11) 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 \( N \to \infty \), (11) becomes equivalent to the solution of our infinite horizon problem, which below we shall show can be expressed as

\[
c(L)y_t = c(\beta L^{-1})^{-1} a_t ,
\]

so that as \( N \to \infty \) we expect that for each fixed \( t \), \( U_{t,t-j}^{-1} \to c_j \) and \( L_{t,t+j} \) approaches the coefficient on \( L^{-j} \) in the expansion of \( c(\beta L^{-1})^{-1} \).

This suspicion is true under general conditions that we shall study later.

For now, we note that by creating the matrix \( W \) for large \( N \) and factoring it into the \( LU \) form, good approximations to \( c(L) \) and \( c(\beta L^{-1})^{-1} \) can be obtained.

### 58.5 The Infinite Horizon Limit

For the infinite horizon problem, we propose to discover first-order necessary conditions by taking the limits of (4) and (5) as \( N \to \infty \).

This approach is valid, and the limits of (4) and (5) as \( N \) approaches infinity are first-order necessary conditions for a maximum.

However, for the infinite horizon problem with \( \beta < 1 \), the limits of (4) and (5) are, in general, not sufficient for a maximum.

That is, the limits of (5) do not provide enough information uniquely to determine the solution of the Euler equation (4) that maximizes (1).

As we shall see below, a side condition on the path of \( y_t \) that together with (4) is sufficient for an optimum is

\[
\sum_{t=0}^{\infty} \beta^t h y_t^2 < \infty
\]

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 (1) than does the optimal path selected by the solution procedure below.

Consider the characteristic equation associated with the Euler equation

\[ h + d (\beta z^{-1}) \frac{d}{dz} = 0 \]  

(13)

Notice that if \( \tilde{z} \) is a root of equation (13), then so is \( \beta \tilde{z}^{-1} \).

Thus, the roots of (13) come in “\( \beta \)-reciprocal” pairs.

Assume that the roots of (13) 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 (13) can be expressed as

\[ h + d (\beta z^{-1}) \frac{d}{d} = z^{-m} z_0 (z - z_1) \cdots (z - z_m) (z - z_{m+1}) \cdots (z - z_{2m}) \]  

(14)

where \( z_0 \) is a constant.

In (14), we substitute \( (z - z_j) = -z_j (1 - \frac{1}{z_j} z) \) and \( (z - \beta z_j^{-1}) = z (1 - \frac{1}{z_j} \beta z^{-1}) \) for \( j = 1, \ldots, m \) to get

\[ h + d (\beta z^{-1}) \frac{d}{d} = (-1)^m (z_0 z_1 \cdots z_m) (1 - \frac{1}{z_1} z) \cdots (1 - \frac{1}{z_m} z) (1 - \frac{1}{z_1} \beta z^{-1}) \cdots (1 - \frac{1}{z_m} \beta z^{-1}) \]  

Now define \( c(z) = \sum_{j=0}^{m} c_j z^j \) as

\[ c(z) = \left[ (-1)^m z_0 z_1 \cdots z_m \right]^{1/2} \left( 1 - \frac{z}{z_1} \right) \left( 1 - \frac{z}{z_2} \right) \cdots \left( 1 - \frac{z}{z_m} \right) \]  

(15)

Notice that (14) can be written

\[ h + d (\beta z^{-1}) \frac{d}{d} = c(\beta z^{-1}) c(z) \]  

(16)

It is useful to write (15) as

\[ c(z) = c_0 (1 - \lambda_1 z) \cdots (1 - \lambda_m z) \]  

(17)

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 \( j = 1, \ldots, m \).
CHAPTER 58. CLASSICAL CONTROL WITH LINEAR ALGEBRA

Using (17), we can express the factorization (16) 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 (16) 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 (16), 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 (12) is

\[ c(L) y_t = c(\beta L^{-1})^{-1} a_t \] (18)

This can be established by using an argument paralleling that in chapter IX of [94].

To exhibit the solution in a form paralleling that of [94], we use (17) to write (18) as

\[ (1 - \lambda_1 L) \cdots (1 - \lambda_m L) y_t = \frac{c_0^{-2} a_t}{(1 - \beta \lambda_1 L^{-1}) \cdots (1 - \beta \lambda_m L^{-1})} \] (19)

Using partial fractions, we can write the characteristic polynomial on the right side of (19) as

\[ \sum_{j=1}^{m} \frac{A_j}{1 - \lambda_j \beta L^{-1}} \quad \text{where} \quad A_j := \frac{c_0^{-2}}{\prod_{i \neq j}(1 - \lambda_i / \lambda_j)} \]

Then (19) 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

\[ (1 - \lambda_1 L) \cdots (1 - \lambda_m L) y_t = \sum_{j=1}^{m} A_j \sum_{k=0}^{\infty} (\lambda_j \beta)^k a_{t+k} \] (20)

Equation (20) 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, (20) is the unique solution of the Euler equation that satisfies the initial conditions and condition (12).

In effect, condition (12) compels us to solve the “unstable” roots of \( h + d(\beta z^{-1})d(z) \) forward (see [94]).

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 (20)
• 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 (20) 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, (20) 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 (12) is satisfied.

In fact, $\max_i |\lambda_i| < 1/\sqrt{\beta}$ is a necessary condition for (12) to hold.

Were (12) 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 (17).

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 (1).

Therefore it is easy to dominate a path that violates (12).

58.6 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 (16) becomes

$$\left( h + d( z^{-1})d(z) \right) = c(z^{-1}) c(z)$$

where

$$c(z) = c_0 (1 - \lambda_1 z) \ldots (1 - \lambda_m z)$$

$$c_0 = \left[(-1)^m z_0 z_1 \ldots z_m \right]$$

$$|\lambda_j| < 1 \text{ for } j = 1, \ldots, m$$

$$\lambda_j = \frac{1}{z_j} \text{ for } j = 1, \ldots, m$$

$$z_0 = \text{ constant}$$

The solution of the problem becomes
\[(1 - \lambda_1 L) \cdots (1 - \lambda_m L)y_t = \sum_{j=1}^{m} A_j \sum_{k=0}^{\infty} \lambda_j^k a_{t+k}\]

### 58.6.1 Transforming discounted to undiscounted problem

Discounted problems can always be converted into undiscounted problems via a simple transformation.

Consider problem (1) with \(0 < \beta < 1\).

Define the transformed variables

\[
\tilde{a}_t = \beta^{t/2} a_t, \quad \tilde{y}_t = \beta^{t/2} y_t
\]

Then notice that \(\beta^t \left| d(L) y_t \right|^2 = \left| \tilde{d}(L) \tilde{y}_t \right|^2\) 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 (1) is equivalent to

\[
\lim_{N \to \infty} \sum_{t=0}^{N} \left\{ \tilde{a}_t \tilde{y}_t - \frac{1}{2} h \tilde{y}_t^2 - \frac{1}{2} [\tilde{d}(L) \tilde{y}_t]^2 \right\}
\]

which is to be maximized over sequences \(\{\tilde{y}_t, t = 0, \ldots\}\) subject to \(\tilde{y}_{-1}, \ldots, \tilde{y}_{-m}\) given and \(\{\tilde{a}_t, t = 1, \ldots\}\) a known bounded sequence.

The Euler equation for this problem is \([h + \tilde{d}(L^{-1}) \tilde{d}(L)] \tilde{y}_t = \tilde{a}_t\).

The solution is

\[(1 - \tilde{\lambda}_1 L) \cdots (1 - \tilde{\lambda}_m L) \tilde{y}_t = \sum_{j=1}^{m} \tilde{A}_j \sum_{k=0}^{\infty} \tilde{\lambda}_j^k \tilde{a}_{t+k}\]

or

\[
\tilde{y}_t = \tilde{f}_1 \tilde{y}_{t-1} + \cdots + \tilde{f}_m \tilde{y}_{t-m} + \sum_{j=1}^{m} \tilde{A}_j \sum_{k=0}^{\infty} \tilde{\lambda}_j^k \tilde{a}_{t+k},
\]

where \(\tilde{c}(z^{-1}) \tilde{c}(z) = h + \tilde{d}(z^{-1}) \tilde{d}(z)\), and where

\[
[(1)^m \tilde{z}_0 \tilde{z}_1 \ldots \tilde{z}_m]^{1/2} (1 - \tilde{\lambda}_1 z) \cdots (1 - \tilde{\lambda}_m z) = \tilde{c}(z), \quad \text{where } |\tilde{\lambda}_j| < 1
\]

We leave it to the reader to show that (23) implies the equivalent form of the solution

\[y_t = f_1 y_{t-1} + \cdots + f_m y_{t-m} + \sum_{j=1}^{m} A_j \sum_{k=0}^{\infty} (\lambda_j^j) a_{t+k}\]

where

\[f_j = \tilde{f}_j \beta^{-j/2}, \quad A_j = \tilde{A}_j, \quad \lambda_j = \tilde{\lambda}_j \beta^{-1/2}\]
The transformations (21) and the inverse formulas (24) allow us to solve a discounted problem by first solving a related undiscounted problem.

58.7 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

In [3]: function LQFilter(d, h, y_m;
    r = nothing,
    β = nothing,
    h_eps = 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 \( \dot{\Phi} \) up front
    ϕ = zeros(2m + 1)
    for i in -m:m
        ϕ[m-i+1] = sum(diag(d'*d', -i))
    end
    ϕ[m+1] = ϕ[m+1] + h

    # if r is given calculate the vector \( \Phi_r \)
    if isnothing(r)
        k = nothing
        ϕ_r = nothing
    else
        k = size(r, 1) - 1
        ϕ_r = zeros(2k + 1)

        for i = -k:k
            ϕ_r[k-i+1] = sum(diag(r'*r', -i))
        end

        if h_eps != nothing
            ϕ_r[k+1] = ϕ_r[k+1] + h_eps
        end
    end

    # if \( \beta \) is given, define the transformed variables
    if isnothing(β)
        β = 1.0
    else
        d = β.^((collect(0:m)/2) * d)
        y_m = y_m * β.^(collect(1:m)/2)
    end

    return (d = d, h = h, y_m = y_m, m = m, ϕ = ϕ, β = β, ϕ_r = ϕ_r, k = k)
end

function construct_W_and_Wm(lqf, N)
@unpack d, m = lqf
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' - Diagonal(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)
@unpack ϕ, h = lqf

W[1:(m + 1), 1:(m + 1)] = D_m1 + h * I
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)] = ϕ'
end

for i in 1:m
    W[N - m + i + 1, end-(2m + 1 - i)*1:end] = ϕ[1:end-i]
end

for i in 1:m
    W_m[N - i + 2, 1:(m - i)+1] = ϕ[(m + 1 + i):end]
end

return W, W_m
end

function roots_of_characteristic(lqf)
    @unpack m, ϕ = lqf

    # Calculate the roots of the 2m-polynomial
    ϕ_poly=Polynomial(ϕ[end:-1:1])
    proots = roots(ϕ_poly)

    # sort the roots according to their length (in descending order)
    roots_sorted = sort(proots, by=abs)[end:-1:1]
end
z_0 = sum(ϕ) / (fromroots(proots))(1.0)  # we need only those outside the unit circle
λ = 1 ./ z_1_to_m
return z_1_to_m, z_0, λ

function coeffs_of_c(lqf)
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

return c_coeffs
end

function solution(lqf)
z_1_to_m, z_0, λ = roots_of_characteristic(lqf)
c_0 = coeffs_of_c(lqf)

A = zeros(lqf.m)
for j in 1:m
denom = 1 - λ/λ[j]
A[j] = c_0^(-2) / prod(denom[1:m .!= j])
end

return λ, A
end

function construct_V(lqf; N=nothing)
@unpack ϕ_r, k = lqf
V = zeros(N, N)
for i in 1:N
for j in 1:N
if abs(i-j) <= k
V[i, j] = ϕ_r[k + abs(i-j)+1]
end
end
return V
end

function simulate_a(lqf, N)
V = construct_V(lqf, N + 1)
d = MVNSampler(zeros(N + 1), V)
return rand(d)
end

function predict(lqf, a_hist, t)
N = length(a_hist) - 1
V = construct_V(lqf, N + 1)

aux_matrix = zeros(N + 1, N + 1)
aux_matrix[1:t+1 , 1:t+1] .= I + zeros(t+1, t+1)
L = chol(V)'
Ea_hist = inv(L) * aux_matrix * L * a_hist

return Ea_hist
end

function optimal_y(lqf, a_hist, t = nothing)
@unpack β, y_m, m = lqf

N = length(a_hist) - 1
W, W_m = construct_W_and_Wm(lqf, N)

F = lu(W)
L, U = F.L, F.U
D = Diagonal(1.0 ./ diag(U))
U = D * U
L = L * Diagonal(1.0 ./ diag(D))

J = reverse(I + zeros(N+1, N + 1), dims = 2)

if isnothing(t)  # if the problem is deterministic
    a_hist = J * a_hist
    # transform the a sequence if β is given
    if β != 1
        a_hist = reshape(a_hist * (β^(collect(N:0)/ 2)), N + 1, 1)
    end
    ā = a_hist - W_m * y_m  # ā from the lecture
    Uy = \( L, ā \)  # U @ ā = L^{-1}ā from the lecture
    ĭ = \( U, Uy \)  # ĭ = U^{-1}L^{-1}ā
    # Revers the order of ĭ with the matrix J
    J = reverse(I + zeros(N+1, N + m + 1), dims = 2)
    y_hist = J * vcat(į, y_m)  # y_hist : concatenated y_m and ĭ
    # transform the optimal sequence back if β is given
    if β != 1
        y_hist = y_hist .* β.^(collect(-m:N)/2)
    end
else  # if the problem is stochastic and we look at it
    Ea_hist = reshape(predict(lqf, a_hist, t), N + 1, 1)
    Ea_hist = J * Ea_hist
    ā = Ea_hist - W_m * y_m  # ā from the lecture
    Uy = \( L, ā \)  # U @ ā = L^{-1}ā from the lecture
    ĭ = \( U, Uy \)  # ĭ = U^{-1}L^{-1}ā
    # Reverse the order of ĭ with the matrix J
    J = reverse(I + zeros(N + m + 1, N + m + 1), dims = 2)
    y_hist = J * vcat(į, y_m)  # y_hist : concatenated y_m and ĭ
    return y_hist, L, U, ĭ
end

Out[3]: optimal_y (generic function with 2 methods)

58.7.1 Example

In this application we’ll have one lag, with
\[ d(L)y_t = \gamma(I - L)y_t = \gamma(y_t - y_{t-1}) \]

Suppose for the moment that \( \gamma = 0 \).

Then the intertemporal component of the LQ problem disappears, and the agent simply wants to maximize \( a_t y_t - hy_t^2 / 2 \) in each period.

This means that the agent chooses \( y_t = a_t / h \).

In the following we’ll set \( h = 1 \), so that the agent just wants to track the \( \{a_t\} \) process.

However, as we increase \( \gamma \), the agent gives greater weight to a smooth time path.

Hence \( \{y_t\} \) evolves as a smoothed version of \( \{a_t\} \).

The \( \{a_t\} \) sequence we’ll choose as a stationary cyclic process plus some white noise.

Here’s some code that generates a plot when \( \gamma = 0.8 \)

```julia
In [4]: gr(fmt=:png);

# set seed and generate a_t sequence
Random.seed!(123)
n = 100
a_seq = sin.(range(0, 5 * pi, length = 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
plt = plot(time, a_seq / h, lw=2, color=:black, alpha=0.8, marker = :circle,
          markersize = 2, label="a_t")
plot!(plt, time, y, lw=2, color=:blue, marker = :circle, markersize = 2,
      alpha=0.8,
      label="y_t")
plot!(plt, xlabel="Time", grid=true, xlim=(0,maximum(time)), legend=:bottomleft)
end

plot_simulation()
```

Out[4]:
Here’s what happens when we change $\gamma$ to 5.0

**In [5]:**  plot_simulation($\gamma=5.0$)

**Out[5]:**

And here’s $\gamma = 10$
58.8 Exercises

58.8.1 Exercise 1

Consider solving a discounted version ($\beta < 1$) of problem (1), as follows.

Convert (1) to the undiscounted problem (22).

Let the solution of (22) in feedback form be

$$(1 - \tilde{\lambda}_1 L) \cdots (1 - \tilde{\lambda}_m L) \tilde{y}_t = \sum_{j=1}^{m} \tilde{A}_j \sum_{k=0}^{\infty} \tilde{\lambda}_j^k \tilde{a}_{t+k}$$

or

$$\tilde{y}_t = \tilde{f}_1 \tilde{y}_{t-1} + \cdots + \tilde{f}_m \tilde{y}_{t-m} + \sum_{j=1}^{m} \tilde{A}_j \sum_{k=0}^{\infty} \tilde{\lambda}_j^k \tilde{a}_{t+k}$$

(25)

Here

- $h + \dd(z^{-1}) \dd(z) = \tilde{c}(z^{-1}) \dd(z)$
- $\dd(z) = [(-1)^m \tilde{z}_0 \tilde{z}_1 \cdots \tilde{z}_m]^{1/2} (1 - \tilde{\lambda}_1 z) \cdots (1 - \tilde{\lambda}_m z)$

where the $\tilde{z}_j$ are the zeros of $h + \dd(z^{-1}) \dd(z)$.

Prove that (25) 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 $\lambda_j = \tilde{\lambda}_j \beta^{-1/2}$.

58.8.2 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” (20), giving numerical values for the coefficients.

Make sure that the boundary conditions (5) are satisfied.

(Note: this problem differs from the problem in the text in one important way: instead of $h > 0$ in (1), $h = 0$. This has an important influence on the solution.)

58.8.3 Exercise 3

Solve the infinite time optimal control problem to maximize

$$\lim_{N \to \infty} \sum_{t=0}^{N} - \frac{1}{2}[(1 - 2L)y_t]^2,$$

subject to $y_{-1}$ given. Prove that the solution is

$$y_t = 2y_{t-1} = 2^{t+1}y_{-1} \quad t > 0$$

58.8.4 Exercise 4

Solve the infinite time problem, to maximize

$$\lim_{N \to \infty} \sum_{t=0}^{N} (.0000001) y_t^2 - \frac{1}{2}[(1 - 2L)y_t]^2$$

subject to $y_{-1}$ given. Prove that the solution $y_t = 2y_{t-1}$ violates condition (12), and so is not optimal.

Prove that the optimal solution is approximately $y_t = .5y_{t-1}$. 
Chapter 59

Classical Filtering With Linear Algebra

59.1 Contents

- Overview 59.2
- Infinite Horizon Prediction and Filtering Problems 59.3
- Finite Dimensional Prediction 59.4
- Combined Finite Dimensional Control and Prediction 59.5
- Exercises 59.6

59.2 Overview

This is a sequel to the earlier lecture Classical Control with Linear Algebra.

That lecture used linear algebra – in particular, the LU decomposition – to formulate and solve a class of linear-quadratic optimal control problems.

In this lecture, we’ll be using a closely related decomposition, the Cholesky decomposition, to solve linear prediction and filtering problems.

We exploit the useful fact that there is an intimate connection between two superficially different classes of problems:

- deterministic linear-quadratic (LQ) optimal control problems
- linear least squares prediction and filtering problems

The first class of problems involves no randomness, while the second is all about randomness. Nevertheless, essentially the same mathematics solves both type of problem.

This connection, which is often termed “duality,” is present whether one uses “classical” or “recursive” solution procedures.

In fact we saw duality at work earlier when we formulated control and prediction problems recursively in lectures LQ dynamic programming problems, A first look at the Kalman filter, and The permanent income model.

A useful consequence of duality is that

- With every LQ control problem there is implicitly affiliated a linear least squares pre-
diction 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 [94] [chs. IX, XIV] and Hansen and Sargent [44] formulated and solved control and filtering problems using z-transform methods.
In this lecture we investigate these ideas using mostly elementary linear algebra.

59.2.1 References

Useful references include [107], [44], [82], [6], and [80].

59.2.2 Setup

In [1]: using InstantiateFromURL
# optionally add arguments to force installation: instantiate = true, "precompile = true"
github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

In [2]: using LinearAlgebra, Statistics

59.3 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,

$$Y_t = d(L)u_t$$  \hspace{1cm} (1)

where $d(L) = \sum_{j=0}^{m} d_j L^j$, and $u_t$ is a serially uncorrelated stationary random process satisfying

$$\mathbb{E}u_t = 0$$

$$\mathbb{E}u_t u_s = \begin{cases} 1 & \text{if } t = s \\ 0 & \text{otherwise} \end{cases}$$  \hspace{1cm} (2)

We impose no conditions on the zeros of $d(z)$.
A second covariance stationary process is $X_t$ given by

$$X_t = Y_t + \varepsilon_t$$  \hspace{1cm} (3)

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 $\mathbb{E}\varepsilon_t u_s = 0$ for all $t$ and $s$. 

59.3. INFINITE HORIZON PREDICTION AND FILTERING PROBLEMS

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 \( \gamma_j(L) = \sum_{k=0}^{\infty} \gamma_{jk} L^k \) 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 \( b(L) = \sum_{j=0}^{\infty} b_j L^j \) 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 [80].

59.3.1 Problem formulation

These problems are solved as follows.

The covariograms of \( Y \) and \( X \) and their cross covariogram are, respectively,

\[
\begin{align*}
C_X(\tau) &= \mathbb{E}X_tX_{t-\tau} \\
C_Y(\tau) &= \mathbb{E}Y_tY_{t-\tau} \quad \tau = 0, \pm 1, \pm 2, \ldots \\
C_{Y,X}(\tau) &= \mathbb{E}Y_tX_{t-\tau}
\end{align*}
\]

The covariance and cross covariance generating functions are defined as

\[
\begin{align*}
g_X(z) &= \sum_{\tau=-\infty}^{\infty} C_X(\tau)z^\tau \\
g_Y(z) &= \sum_{\tau=-\infty}^{\infty} C_Y(\tau)z^\tau \\
g_{Y,X}(z) &= \sum_{\tau=-\infty}^{\infty} C_{Y,X}(\tau)z^\tau
\end{align*}
\]

The generating functions can be computed by using the following facts.

Let \( v_{1t} \) and \( v_{2t} \) be two mutually and serially uncorrelated white noises with unit variances.

That is, \( \mathbb{E}v_{1t}^2 = \mathbb{E}v_{2t}^2 = 1, \mathbb{E}v_{1t} = \mathbb{E}v_{2t} = 0, \mathbb{E}v_{1t}v_{2s} = 0 \) for all \( t \) and \( s, \mathbb{E}v_{1t}v_{1t-j} = \mathbb{E}v_{2t}v_{2t-j} = 0 \) for all \( j \neq 0 \).

Let \( x_t \) and \( y_t \) be two random process given by

\[
\begin{align*}
y_t &= A(L)v_{1t} + B(L)v_{2t} \\
x_t &= C(L)v_{1t} + D(L)v_{2t}
\end{align*}
\]

Then, as shown for example in [94] [ch. XI], it is true that

\[
\begin{align*}
g_y(z) &= A(z)A(z^{-1}) + B(z)B(z^{-1}) \\
g_x(z) &= C(z)C(z^{-1}) + D(z)D(z^{-1}) \\
g_{yx}(z) &= A(z)C(z^{-1}) + B(z)D(z^{-1})
\end{align*}
\]

Applying these formulas to (1) – (4), we have
\[ g_Y(z) = d(z)d(z^{-1}) \]
\[ g_X(z) = d(z)d(z^{-1}) + h \quad (7) \]
\[ g_{YX}(z) = d(z)d(z^{-1}) \]

The key step in obtaining solutions to our problems is to factor the covariance generating function \( g_X(z) \) of \( X \).

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,

\[ X_t = c(L)\eta_t \quad (8) \]

where \( c(L) = \sum_{j=0}^{m} c_j L^j \), with

\[ c_0 \eta_t = X_t - \hat{\mathbb{E}}[X_t|X_{t-1}, X_{t-2}, ...] \quad (9) \]

Here \( \hat{\mathbb{E}} \) is the linear least squares projection operator.

Equation (9) 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 (9) requires that \( \eta_t \) lie in the closed linear space spanned by \( [X_t, X_{t-1}, ...] \).

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 (9) that \( \eta_t \) is a serially uncorrelated random process, and that a normalization can be imposed so that \( \mathbb{E}\eta_t^2 = 1 \).

Consequently, an implication of (8) is that the covariance generating function of \( X_t \) can be expressed as

\[ g_X(z) = c(z)c(z^{-1}) \quad (10) \]

It remains to discuss how \( c(L) \) is to be computed.

Combining (6) and (10) gives

\[ d(z)d(z^{-1}) + h = c(z)c(z^{-1}) \quad (11) \]

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:

\[ \left[ \sum_{j=-\infty}^{\infty} f_j L^j \right]_+ \equiv \sum_{j=0}^{\infty} f_j L^j \quad (12) \]

In words, \([ \quad ]_+ \) means “ignore negative powers of \( L \).”

We have defined the solution of the prediction problem as \( \hat{\mathbb{E}}[X_{t+j}|X_t, X_{t-1}, ...] = \gamma_j(L)X_t \).

Assuming that the roots of \( c(z) = 0 \) all lie outside the unit circle, the Wiener-Kolmogorov formula for \( \gamma_j(L) \) holds:
\[
\gamma_j(L) = \left[ \frac{c(L)}{L^j} \right] + c(L)^{-1}
\]  \hspace{1cm} (13)

We have defined the solution of the filtering problem as
\[
\hat{\mathbb{E}}[Y_t \mid X_t, X_{t-1}, \ldots] = b(L)X_t.
\]

The Wiener-Kolomogorov formula for \(b(L)\) is
\[
b(L) = \left[ \frac{g_{YX}(L)}{c(L^{-1})} \right] + c(L)^{-1}
\]
or
\[
b(L) = \left[ \frac{d(L)d(L^{-1})}{c(L^{-1})} \right] + c(L)^{-1}
\]  \hspace{1cm} (14)

Formulas (13) and (14) are discussed in detail in [108] and [94].

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 [80].

As an example of the usefulness of formula (14), we let \(X_t\) be a stochastic process with Wold moving average representation

\[
X_t = c(L)\eta_t
\]

where \(\mathbb{E}\eta_t^2 = 1\), and \(c_0\eta_t = X_t - \hat{\mathbb{E}}[X_t|X_{t-1}, \ldots], c(L) = \sum_{j=0}^m c_j L^j\).

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 (14) to obtain the answer.

Using the standard formulas (6), we have that

\[
g_{yx}(z) = (1 - \delta z^{-1})c(z)c(z^{-1})
\]
\[
g_x(z) = c(z)c(z^{-1})
\]

Then (14) becomes

\[
b(L) = \left[ \frac{c(L)}{1 - \delta L^{-1}} \right] + c(L)^{-1}
\]  \hspace{1cm} (15)

In order to evaluate the term in the annihilation operator, we use the following result from [44].

**Proposition** Let

- \(g(z) = \sum_{j=0}^{\infty} g_j z^j\) where \(\sum_{j=0}^{\infty} |g_j|^2 < +\infty\)
- \(h(z^{-1}) = (1 - \delta_1 z^{-1}) \ldots (1 - \delta_n z^{-1})\), where \(|\delta_j| < 1\), for \(j = 1, \ldots, n\)
Then
\[
\left[ \frac{g(z)}{h(z^{-1})} \right]_+ = \frac{g(z)}{h(z^{-1})} - \sum_{j=1}^{n} \frac{\delta_j g(\delta_j)}{\prod_{k \neq j}^{n} (\delta_j - \delta_k)} \left( \frac{1}{z - \delta_j} \right)
\]  
(16)

and, alternatively,
\[
\left[ \frac{g(z)}{h(z^{-1})} \right]_+ = \sum_{j=1}^{n} B_j \left( \frac{zg(z) - \delta_j g(\delta_j)}{z - \delta_j} \right)
\]  
(17)

where \( B_j = 1/\prod_{k=1}^{n} (1 - \delta_k/\delta_j) \).

Applying formula (17) of the proposition to evaluating (15) with \( g(z) = c(z) \) and \( h(z^{-1}) = 1 - \delta z^{-1} \) gives
\[
b(L) = \left[ \frac{Lc(L) - \delta c(\delta)}{L - \delta} \right] c(L)^{-1}
\]
or
\[
b(L) = \left[ \frac{1 - \delta c(\delta)L^{-1}c(L)^{-1}}{1 - \delta L^{-1}} \right]
\]

Thus, we have
\[
\hat{\mathbb{E}} \left[ \sum_{j=0}^{\infty} \delta^j X_{t+j} | X_t, x_{t-1}, \ldots \right] = \left[ 1 - \delta c(\delta)L^{-1}c(L)^{-1} \right] X_t
\]  
(18)

This formula is useful in solving stochastic versions of problem 1 of lecture Classical Control with Linear Algebra in which the randomness emerges because \( \{a_t\} \) is a stochastic process.

The problem is to maximize
\[
\mathbb{E}_t \lim_{N \to \infty} \sum_{t=0}^{N} \beta^t \left[ a_t y_t - \frac{1}{2} h y_t^2 - \frac{1}{2} [d(L)y_t]^2 \right]
\]  
(19)

where \( \mathbb{E}_t \) is mathematical expectation conditioned on information known at \( t \), and where \( \{a_t\} \) is a covariance stationary stochastic process with Wold moving average representation

\[
a_t = c(L) \eta_t
\]

where
\[
c(L) = \sum_{j=0}^{\tilde{n}} c_j L^j
\]

and
\[
\eta_t = a_t - \hat{\mathbb{E}}[a_t | a_{t-1}, \ldots]
\]
59.4. **FINITE DIMENSIONAL PREDICTION**

The problem is to maximize (19) with respect to a contingency plan expressing \( y_t \) as a function of information known at \( t \), which is assumed to be \( (y_{t-1}, y_{t-2}, \ldots, a_t, a_{t-1}, \ldots) \).

The solution of this problem can be achieved in two steps.

First, ignoring the uncertainty, we can solve the problem assuming that \( \{a_t\} \) is a known sequence.

The solution is, from above,

\[
 c(L)y_t = c(\beta L^{-1})^{-1}a_t
\]

or

\[
 (1 - \lambda_1 L) \ldots (1 - \lambda_m L)y_t = \sum_{j=1}^{m} A_j \sum_{k=0}^{\infty} (\lambda_j \beta)^k a_{t+k}
\]  

(20)

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 (18) and (20), we have the following solution

\[
 (1 - \lambda_1 L) \ldots (1 - \lambda_m L)y_t = \sum_{j=1}^{m} A_j \left[ \frac{1 - \beta \lambda_j c(\beta \lambda_j) L^{-1} c(L)^{-1}}{1 - \beta \lambda_j L^{-1}} \right] a_t
\]

59.4 **Finite Dimensional Prediction**

Let \((x_1, x_2, \ldots, x_T) = x\) be a \( T \times 1 \) vector of random variables with mean \( \mathbb{E}x = 0 \) and covariance matrix \( \mathbb{E}xx' = V \).

Here \( V \) is a \( T \times T \) positive definite matrix.

We shall regard the random variables as being ordered in time, so that \( x_t \) is thought of as the value of some economic variable at time \( t \).

For example, \( x_t \) could be generated by the random process described by the Wold representation presented in equation (8).

In this case, \( V_{ij} \) is given by the coefficient on \( z^{|i-j|} \) in the expansion of \( g_x(z) = d(z) d(z^{-1}) + h \), which equals \( h + \sum_{k=0}^{\infty} d_k d_{k+|i-j|} \).

We shall be interested in constructing \( j \) step ahead linear least squares predictors of the form

\[ \hat{E} [x_T|x_{T-j}, x_{T-j+1}, \ldots, x_1] \]

where \( \hat{E} \) 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

\[ L V L' = I \]

where \( L \) is lower-triangular, 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 \( \mathbb{E} \varepsilon \varepsilon' = L \mathbb{E} x x' L' = I \).

It is convenient to write out the equations \( Lx = \varepsilon \) and \( L^{-1} \varepsilon = x \)

\[
\begin{align*}
L_{11}x_1 &= \varepsilon_1 \\
L_{21}x_1 + L_{22}x_2 &= \varepsilon_2 \\
&\vdots \\
L_{T1}x_1 + \ldots + L_{TT}x_T &= \varepsilon_T
\end{align*}
\]

or

\[
\sum_{j=0}^{t-1} L_{t,t-j}x_{t-j} = \varepsilon_t, \quad t = 1, 2, \ldots T
\]  

We also have

\[
x_t = \sum_{j=0}^{t-1} L_{t,t-j}^{-1}\varepsilon_{t-j} \tag{23}
\]

Notice from (23) that \( x_t \) is in the space spanned by \( \varepsilon_t, \varepsilon_{t-1}, \ldots, \varepsilon_1 \), and from (22) that \( \varepsilon_t \) is in the space spanned by \( x_t, x_{t-1}, \ldots, x_1 \).

Therefore, we have that for \( t - 1 \geq m \geq 1 \)

\[
\hat{\mathbb{E}}[x_t \mid x_{t-m}, x_{t-m-1}, \ldots, x_1] = \hat{\mathbb{E}}[x_t \mid \varepsilon_{t-m}, \varepsilon_{t-m-1}, \ldots, \varepsilon_1] \tag{24}
\]

For \( t - 1 \geq m \geq 1 \) rewrite (23) as

\[
x_t = \sum_{j=0}^{m-1} L_{t,t-j}^{-1}\varepsilon_{t-j} + \sum_{j=m}^{t-1} L_{t,t-j}^{-1}\varepsilon_{t-j} \tag{25}
\]

Representation (25) is an orthogonal decomposition of \( x_t \) into a part \( \sum_{j=m}^{t-1} L_{t,t-j}^{-1}\varepsilon_{t-j} \) that lies in the space spanned by \( [x_{t-m}, x_{t-m+1}, \ldots, x_1] \), and an orthogonal component not in this space.

59.4.1 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
In [3]: using Polynomials.PolyCompat, LinearAlgebra
import Polynomials.PolyCompat: roots, coeffs

function LQFilter(d, h, y_m;
    r = nothing, 
    β = nothing, 
    h_eps = 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 \( \phi \) up front
ϕ = zeros(2m + 1)
for i in -m:m
    ϕ[m-i+1] = sum(diag(d'd', -i))
end
ϕ[m+1] = ϕ[m+1] + h

# if \( r \) is given calculate the vector \( \phi_r \)
if isnothing(r)
    k = nothing
    ϕ_r = nothing
else
    k = size(r, 1) - 1
    ϕ_r = zeros(2k + 1)
    for i = -k:k
        ϕ_r[k-i+1] = sum(diag(r'r', -i))
    end
    if isnothing(h_eps) == false
        ϕ_r[k+1] = ϕ_r[k+1] + h_eps
    end
end

# if \( \beta \) is given, define the transformed variables
if isnothing(β)
    β = 1.0
else
    d = β.(collect(0:m)/2) * d
    y_m = y_m * β.(-collect(1:m)/2)
end

return (d = d, h = h, y_m = y_m, m = m, ϕ = ϕ, β = β, ϕ_r = ϕ_r, k = k)
end

function construct_W_and_Wm(lqf, N)

    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)

    return (d = d, h = h, y_m = y_m, m = m, ϕ = ϕ, β = β, ϕ_r = ϕ_r, k = k, W = W, W_m = W_m, D_m1 = D_m1, M = M)
# (1) construct the \(D_{m+1}\) matrix using the formula

```python
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

\(D_m1 = D_m1 + D_m1' - \text{Diagonal}(\text{diag}(D_m1))\)

# (2) Construct the \(M\) matrix using the entries of \(D_m1\)

```python
for j in 1:m
  for i in (j + 1):(m + 1)
    M[i, j] = D_m1[i-j, m+1]
  end
end
```

\(\phi, h = \text{lqf}._m, \text{lqf}._\phi\)

\(W[1:(m + 1), 1:(m + 1)] = D_m1 + h * \text{I}\)
\(W[1:(m + 1), (m + 2):(2m + 1)] = M\)

```python
for (i, row) in enumerate((m + 2):(N + 1 - m))
  W[row, (i + 1):(2m + 1 + i)] = \phi'
end
```

```python
for i in 1:m
  W[N - m + i + 1, end-(2m + 1 - i)+1:end] = \phi[1:end-i]
end
```

```python
for i in 1:m
  W_m[N - i + 2, 1:(m - i)+1] = \phi[(m + 1 + i):end]
end
```

return \(W, W_m\)

```python
function roots_of_characteristic(lqf)
  m, \phi = lqf._m, lqf._\phi

  \(\phi\_\text{poly} = \text{Poly}(\phi[end:-1:1])\)
  \(\text{proots} = \text{roots}(\phi\_\text{poly})\)

  \(\text{roots\_sorted} = \text{sort}(\text{proots}, \text{by}=\text{abs})[\text{end:-1:1}]\)

  \(\text{z\_0 = sum}(\phi) / \text{polyval}(\phi(\text{proots}), 1.0)\)
  \(\text{z\_1\_to\_m = roots\_sorted[1:m]}\)

  # we need only those outside the unit circle

  \(\lambda = 1 .\! /\! z\_1\_to\_m\)

  return z\_1\_to\_m, z\_0, \lambda
end
```

```python
function coeffs_of_c(lqf)
```

```python
```

```python
```

```python
```
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

end

function solution(lqf)
  z_1_to_m, z_0, λ = roots_of_characteristic(lqf)
c_0 = coeffs_of_c(lqf)
  A = zeros(lqf.m)
  for j in 1:m
    denom = 1 - λ/λ[j]
    A[j] = c_0^(-2) / prod(denom[1:m .!= j])
  end
  return λ, A
end

function construct_V(lqf; N = nothing)
  if isnothing(N)
    error("N must be provided!!")
  end
  if !(N isa Integer)
    throw(ArgumentError("N must be Integer!"))
  end
  ϕ_r, k = lqf.ϕ_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] = ϕ_r[k + abs(i-j)+1]
      end
    end
  end
  return V
end

function simulate_a(lqf, N)
  V = construct_V(N + 1)
  d = MVNSampler(zeros(N + 1), V)
  return rand(d)
end

function predict(lqf, a_hist, t)
  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] = Matrix(I, t + 1, t + 1)
  L = cholesky(V).U'
  Ea_hist = inv(L) * aux_matrix * L * a_hist
  return Ea_hist
end

function optimal_y(lqf, a_hist, t = nothing)
  β, y_m, m = lqf.β, lqf.y_m, lqf.m
\( N = \text{length}(a_{\text{hist}}) - 1 \)

\( W, \ W_m = \text{construct}\_W\_\text{and}\_Wm(lqf, N) \)

\( F = \text{lu}(W, \ \text{Val(true)}) \)

\( L, U = F \)

\( D = \text{diagm}(\emptyset \Rightarrow 1.0 \/. \text{diag}(U)) \)

\( U = D \times U \)

\( L = L \times \text{diagm}(\emptyset \Rightarrow 1.0 \/. \text{diag}(D)) \)

\( J = \text{reverse}([I, N + 1, N + 1], \text{dims} = 2) \)

\textbf{if} isnothing(t) \hspace{1em} \# if the problem is deterministic

\hspace{1em} a_{\text{hist}} = J \times a_{\text{hist}}

\hspace{1em} # transform the a sequence if \( \beta \) is given

\hspace{2em} \textbf{if} \( \beta \neq 1 \)

\hspace{3em} a_{\text{hist}} = \text{reshape}(a_{\text{hist}} \times (\beta^{(\text{collect}(N:0)/2)}), N + 1, 1)

\hspace{1em} \text{end}

\hspace{1em} \tilde{a} = a_{\text{hist}} - W_m \times y_m \hspace{1em} \# \tilde{a} \text{ from the lecture}

\hspace{1em} Uy = \langle (L, \tilde{a}) \rangle \hspace{1em} \# U \odot \tilde{y} = L^{-1}\tilde{a} \text{ from the lecture}

\hspace{1em} \hat{y} = \langle (U, Uy) \rangle \hspace{1em} \# \hat{y} = U^{-1}L^{-1}\tilde{a}

\hspace{1em} #\text{Reverse the order of} \hat{y} \text{with the matrix} J

\hspace{1em} J = \text{reverse}([I, N + m + 1, N + m + 1], \text{dims} = 2)

\hspace{1em} y_{\text{hist}} = J \times \text{vcat}(\hat{y}, y_m) \hspace{1em} \# y_{\text{hist}} : \text{concatenated} y_m \text{ and} \hat{y}

\hspace{1em} #\text{transform the optimal sequence back if} \beta \text{ is given}

\hspace{2em} \textbf{if} \( \beta \neq 1 \)

\hspace{3em} y_{\text{hist}} = y_{\text{hist}} \times \beta^{(-\text{collect(-m:N)/2})} 

\hspace{1em} \text{end}

\textbf{else} \hspace{1em} \# if the problem is stochastic and\( \mathbb{E} \)

\hspace{1em} \text{we look at it}

\hspace{2em} E_{a_{\text{hist}}} = \text{reshape}(\text{predict}(a_{\text{hist}}, t), N + 1, 1)

\hspace{2em} E_{a_{\text{hist}}} = J \times E_{a_{\text{hist}}}

\hspace{2em} \tilde{a} = E_{a_{\text{hist}}} - W_m \times y_m \hspace{1em} \# \tilde{a} \text{ from the lecture}

\hspace{2em} Uy = \langle (L, \tilde{a}) \rangle \hspace{1em} \# U \odot \tilde{y} = L^{-1}\tilde{a} \text{ from the lecture}

\hspace{2em} \hat{y} = \langle (U, Uy) \rangle \hspace{1em} \# \hat{y} = U^{-1}L^{-1}\tilde{a}

\hspace{2em} #\text{Reverse the order of} \hat{y} \text{with the matrix} J

\hspace{2em} J = \text{reverse}([I, N + m + 1, N + m + 1], \text{dims} = 2)

\hspace{2em} y_{\text{hist}} = J \times \text{vcat}(\hat{y}, y_m) \hspace{1em} \# y_{\text{hist}} : \text{concatenated} y_m \text{ and} \hat{y}

\hspace{2em} \text{end}

\hspace{1em} \text{return} y_{\text{hist}}, L, U, \hat{y}

\textbf{end}

\textbf{Out[3]:} \text{optimal_y} (\text{generic function with 2 methods})

Let’s use this code to tackle two interesting examples.
59.4.2 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 (13) 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

\[
\begin{align*}
\text{In [4]:} & \quad m = 1 \\
& \quad y_m = \text{zeros}(m) \\
& \quad d = [1.0, -2.0] \\
& \quad r = [1.0, -2.0] \\
& \quad h = 0.0 \\
& \quad \text{example} = \text{LQFilter}(d, h, y_m, r)
\end{align*}
\]

\[
\begin{align*}
\text{Out[4]:} & \quad (d = [1.0, -2.0], h = 0.0, y_m = [0.0], m = 1, \phi = [-2.0, 5.0, -2.0], \beta = [1.0, -2.0, 5.0, -2.0], k = 1)
\end{align*}
\]

The Wold representation is computed by \text{example.coefficients_of_c().}

Let’s check that it “flips roots” as required

\[
\begin{align*}
\text{In [5]:} & \quad \text{coeffs_of_c(example)} \\
\text{Out[5]:} & \quad 2\text{-element Array}\{\text{Float64,1}\}:
\begin{align*}
2.0 \\
-1.0
\end{align*}
\end{align*}
\]

\[
\begin{align*}
\text{In [6]:} & \quad \text{roots_of_characteristic(example)} \\
\text{Out[6]:} & \quad ([2.0], -2.0, [0.5])
\end{align*}
\]

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 \( V = L^{-1}L^{-1} = LiL' \) and use it to form the vector of “moving average representations” \( x = Li\varepsilon \) and the vector of “autoregressive representations” \( Lx = \varepsilon \)

\[
\begin{align*}
\text{In [7]:} & \quad V = \text{construct_V(example,N=5)} \\
\text{Out[7]:} & \quad 5\times5 \text{ Array}\{\text{Float64,2}\}:
\begin{align*}
5.0 & -2.0 & 0.0 & 0.0 & 0.0 \\
-2.0 & 5.0 & -2.0 & 0.0 & 0.0 \\
0.0 & -2.0 & 5.0 & -2.0 & 0.0 \\
0.0 & 0.0 & -2.0 & 5.0 & -2.0 \\
0.0 & 0.0 & 0.0 & -2.0 & 5.0
\end{align*}
\end{align*}
\]
CHAPTER 59. CLASSICAL FILTERING WITH LINEAR ALGEBRA

Notice how the lower rows of the “moving average representations” are converging to the appropriate infinite history Wold representation

In [8]: F = cholesky(V)
   Li = F.L

Out[8]: 5x5 LowerTriangular{Float64,Array{Float64,2}}:
   2.23607 ippipp
   -0.89442 2.04939 ippipp
   0.0 -0.9759 2.01187 ippipp
   0.0 0.8 -0.9941 2.00294 ippipp
   0.0 0.8 0.0 -0.99853 2.00073

Notice how the lower rows of the “autoregressive representations” are converging to the appropriate infinite history autoregressive representation

In [9]: L = inv(Li)

Out[9]: 5x5 LowerTriangular{Float64,Array{Float64,2}}:
   0.447214 ippipp
   0.19518 0.48795 ippipp
   0.0946762 0.236691 0.49705 ippipp
   0.0469989 0.117474 0.246696 0.499266 ippipp
   0.0234518 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.

59.4.3 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 (13) to compute the linear least squares forecasts $\hat{E}[X_{t+j} \mid X_{t-1}, \ldots]$ for $j = 1, 2, 3$.

We proceed in the same way as example 1.

In [10]:
\[
\begin{align*}
m &= 2 \\
y_m &= [0.0, 0.0] \\
d &= [1.0, 0.0, -\text{sqrt}(2)] \\
r &= [1.0, 0.0, -\text{sqrt}(2)] \\
h &= 0.0 \\
\text{example} &= \text{LQFilter}(d, h, y_m, r = d)
\end{align*}
\]

Out[10]:
\[
\begin{align*}
d &= [1.0, 0.0, -1.4142135623730951], \\
h &= 0.0, \\
y_m &= [0.0, 0.0], \\
m &= 2, \\
\phi &= [-1.4142135623730951, 0.0, 3.0000000000000004, 0.0, -1.4142135623730951], \\
\beta &= 1.0, \\
\phi_r &= [-1.4142135623730951, 0.0, 3.0000000000000004, 0.0, -1.4142135623730951], \\
k &= 2
\end{align*}
\]

In [11]:
\[
\begin{align*}
\text{coeffs_of_c(example)}
\end{align*}
\]

Out[11]:
\[
3\text{-element Array\{Float64,1\}:} \\
1.4142135623731003 \\
-0.0 \\
-1.0000000000000064
\]

In [12]:
\[
\begin{align*}
\text{roots_of_characteristic(example)}
\end{align*}
\]

Out[12]:
\[
(\begin{array}{c}
1.1892071150027195, -1.1892071150027195 \\
-1.4142135623731096, 0.8408964152537157, -0.8408964152537157
\end{array})
\]

In [13]:
\[
V = \text{construct}_V(\text{example, } N = 8)
\]

Out[13]:
\[
8\times8 \text{ Array\{Float64,2\}:} \\
3.0 0.0 -1.4142 0.0 ... 0.0 0.0 0.0 \\
0.0 3.0 0.0 -1.4142 0.0 0.0 0.0 \\
-1.4142 0.0 3.0 0.0 0.0 0.0 0.0 \\
0.0 -1.4142 0.0 3.0 -1.4142 0.0 0.0 \\
0.0 0.0 -1.4142 0.0 0.0 -1.4142 0.0 \\
0.0 0.0 0.0 -1.4142 ... 3.0 0.0 -1.4142 \\
0.0 0.0 0.0 0.0 0.0 3.0 0.0 \\
0.0 0.0 0.0 0.0 -1.4142 0.0 3.0
\]

In [14]:
\[
F = \text{cholesky}(V) \\
L_i = F.L \\
L_i[end-2:end, :]
\]

Out[14]:
\[
3\times8 \text{ Array\{Float64,2\}:} \\
0.0 0.0 0.0 -0.92582 0.0 1.46385 0.0 0.0 \\
0.0 0.0 0.0 0.0 -0.966092 0.0 1.43759 0.0 \\
0.0 0.0 0.0 0.0 0.0 -0.966092 0.0 1.43759
\]

In [15]:
\[
L = \text{inv}(L_i)
\]
59.4.4 Prediction

It immediately follows from the “orthogonality principle” of least squares (see [6] or [94] [ch. X]) that

\[
\hat{E}[x_t | x_{t-m}, x_{t-m+1}, \ldots, x_1] = \sum_{j=m}^{t-1} L_{t,j}^{-1} e_{t-j}
\]

\[
= [L_{t,1}^{-1} L_{t,2}^{-1}, \ldots, L_{t,t-m}^{-1} 0 0 \ldots 0] L x
\]

(26)

This can be interpreted as a finite-dimensional version of the Wiener-Kolmogorov m-step ahead prediction formula.

We can use (26) to represent the linear least squares projection of the vector \(x\) conditioned on the first \(s\) observations \([x_s, x_{s-1}, \ldots, x_1]\).

We have

\[
\hat{E}[x | x_s, x_{s-1}, \ldots, x_1] = L^{-1} \begin{bmatrix} I_s & 0 \\ 0 & 0_{(t-s)} \end{bmatrix} L x
\]

(27)

This formula will be convenient in representing the solution of control problems under uncertainty.

Equation (23) can be recognized as a finite dimensional version of a moving average representation.

Equation (22) can be viewed as a finite dimension version of an autoregressive representation.

Notice that even if the \(x_t\) process is covariance stationary, so that \(V\) is such that \(V_{ij}\) depends only on \(|i - j|\), the coefficients in the moving average representation are time-dependent, there being a different moving average for each \(t\).

If \(x_t\) is a covariance stationary process, the last row of \(L^{-1}\) converges to the coefficients in the Wold moving average representation for \(\{x_t\}\) as \(T \to \infty\).

Further, if \(x_t\) is covariance stationary, for fixed \(k\) and \(j > 0\), \(L_{T-k,T-j}^{-1}\) converges to \(L_{t-k,t-j}^{-1}\) as \(T \to \infty\).

That is, the “bottom” rows of \(L^{-1}\) converge to each other and to the Wold moving average coefficients as \(T \to \infty\).

This last observation gives one simple and widely-used practical way of forming a finite \(T\) approximation to a Wold moving average representation.

First, form the covariance matrix \(\mathbb{E}xx' = V\), then obtain the Cholesky decomposition \(L^{-1}L^{-1'}\) of \(V\), which can be accomplished quickly on a computer.
The last row of $L^{-1}$ gives the approximate Wold moving average coefficients. This method can readily be generalized to multivariate systems.

59.5 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 $d(L) = d_{0} + d_{1}L + ... + d_{m}L^{m}$, $L$ is the lag operator, $\tilde{a} = [a_{N}, a_{N-1}, ..., a_{1}, a_{0}]'$ a random vector with mean zero and $\mathbb{E} \tilde{a}\tilde{a}' = V$.

The variables $y_{-1}, ..., y_{-m}$ are given. Maximization is over choices of $y_{0}, y_{1}, ..., y_{N}$, 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

$$
U\tilde{y} = L^{-1} \tilde{a} + K \begin{bmatrix} y_{-1} \\ \vdots \\ y_{-m} \end{bmatrix}
$$

for some $(N + 1) \times m$ matrix $K$.

Using a version of formula (26), we can express $\hat{\mathbb{E}}[\tilde{a} | a_{s}, a_{s-1}, ..., a_{0}]$ as

$$
\hat{\mathbb{E}}[\tilde{a} | a_{s}, a_{s-1}, ..., a_{0}] = \tilde{U}^{-1} \begin{bmatrix} 0 & 0 \\ 0 & I_{(s+1)} \end{bmatrix} \tilde{U} \tilde{a}
$$

where $I_{(s+1)}$ is the $(s + 1) \times (s + 1)$ identity matrix, and $V = \tilde{U}^{-1} \tilde{U}^{-1}'$, where $\tilde{U}$ is the upper triangular Cholesky factor of the covariance matrix $V$.

(We have reversed the time axis in dating the $a$’s relative to earlier)

The time axis can be reversed in representation (27) 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

$$
U\tilde{y} = L^{-1} \tilde{U}^{-1} \begin{bmatrix} 0 & 0 \\ 0 & I_{(t+1)} \end{bmatrix} \tilde{U} \tilde{a} + K \begin{bmatrix} y_{-1} \\ \vdots \\ y_{-m} \end{bmatrix}
$$

59.6 Exercises

59.6.1 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

\[ \mathbb{E} \tilde{X}_{t+1} | X_t, X_{t-1}, ... = \sum_{j=0}^{\infty} A_{1j} X_{t-j} \]

Find a formula for the \( A_{2j} \)'s in

\[ \mathbb{E} \tilde{X}_{t+2} | X_t, X_{t-1}, ... = \sum_{j=0}^{\infty} A_{2j} X_{t-j} \]

**59.6.2 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 \( t \), \( \mathbb{E} U_t U_s' = 0 \) for all \( s \neq t \), and \( \mathbb{E} U_t U_t' = I \) for all \( t \).

Let \( \varepsilon_t \) be an \( n \times 1 \) vector white noise with mean 0 and contemporaneous covariance matrix \( H \), where \( H \) is a positive definite matrix.

Let \( X_t = Y_t + \varepsilon_t \).

Define the covariograms as \( C_X(\tau) = \mathbb{E} X_t X_{t-\tau}' \), \( C_Y(\tau) = \mathbb{E} Y_t Y_{t-\tau}' \), \( C_{YX}(\tau) = \mathbb{E} Y_t X_{t-\tau}' \).

Then define the matrix covariance generating function, as in (21), only interpret all the objects in (21) as matrices.

Show that the covariance generating functions are given by

\[
\begin{align*}
  g_y(z) &= D(z)D(z^{-1})' \\
  g_X(z) &= D(z)D(z^{-1})' + H \\
  g_{YX}(z) &= D(z)D(z^{-1})'
\end{align*}
\]

A factorization of \( g_X(z) \) can be found (see [91] or [108]) 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, \( X_t - \hat{E}[X_t | X_{t-1}, X_{t-2} ...] = C_0 \eta_t \).

The optimum predictor of \( X_{t+j} \) is

\[ \hat{E}[X_{t+j} | X_t, X_{t-1}, ...] = \left[ \frac{C(L)}{L^j} \right]_+ \eta_t \]

If \( C(L) \) is invertible, i.e., if the zeros of det \( C(z) \) lie strictly outside the unit circle, then this formula can be written

\[ \hat{E}[X_{t+j} | X_t, X_{t-1}, ...] = \left[ \frac{C(L)}{L^j} \right]_+ C(L)^{-1} X_t \]
Part VIII

Dynamic Programming Squared
Chapter 60

Dynamic Stackelberg Problems

60.1 Contents

- Duopoly 60.2
- The Stackelberg Problem 60.3
- Stackelberg Plan 60.4
- Recursive Representation of Stackelberg Plan 60.5
- Computing the Stackelberg Plan 60.6
- Exhibiting Time Inconsistency of Stackelberg Plan 60.7
- Recursive Formulation of the Follower’s Problem 60.8
- Markov Perfect Equilibrium 60.9
- MPE vs. Stackelberg 60.10

This notebook formulates and computes a plan that a Stackelberg leader uses to manipulate forward-looking decisions of a Stackelberg follower that depend on continuation sequences of decisions made once and for all by the Stackelberg leader at time 0.

To facilitate computation and interpretation, we formulate things in a context that allows us to apply linear optimal dynamic programming.

From the beginning we carry along a linear-quadratic model of duopoly in which firms face adjustment costs that make them want to forecast actions of other firms that influence future prices.

60.2 Duopoly

Time is discrete and is indexed by \( t = 0, 1, \ldots \).

Two firms produce a single good whose demand is governed by the linear inverse demand curve

\[
p_t = a_0 - a_1(q_{1t} + q_{2t})
\]

where \( q_{it} \) is output of firm \( i \) at time \( t \) and \( a_0 \) and \( a_1 \) are both positive.

\( q_{10}, q_{20} \) are given numbers that serve as initial conditions at time 0.

By incurring a cost of change
where $\gamma > 0$, firm $i$ can change its output according to

$$q_{it+1} = q_{it} + v_{it}$$

Firm $i$’s profits at time $t$ equal

$$\pi_{it} = p_t q_{it} - \gamma v_{it}^2$$

Firm $i$ wants to maximize the present value of its profits

$$\sum_{t=0}^{\infty} \beta^t \pi_{it}$$

where $\beta \in (0, 1)$ is a time discount factor.

### 60.2.1 Stackelberg Leader and Follower

Each firm $i = 1, 2$ chooses a sequence $\vec{q}_i \equiv \{q_{it+1}\}_{i=0}^{\infty}$ once and for all at time 0.

We let firm 2 be a **Stackelberg leader** and firm 1 be a **Stackelberg follower**.

The leader firm 2 goes first and chooses $\{q_{2t+1}\}_{t=0}^{\infty}$ once and for all at time 0.

Knowing that firm 2 has chosen $\{q_{2t+1}\}_{t=0}^{\infty}$, the follower firm 1 goes second and chooses $\{q_{1t+1}\}_{t=0}^{\infty}$ once and for all at time 0.

In choosing $\vec{q}_2$, firm 2 takes into account that firm 1 will base its choice of $\vec{q}_1$ on firm 2’s choice of $\vec{q}_2$.

### 60.2.2 Abstract Statement of the Leader’s and Follower’s Problems

We can express firm 1’s problem as

$$\max_{\vec{q}_1} \Pi_1(\vec{q}_1; \vec{q}_2)$$

where the appearance behind the semi-colon indicates that $\vec{q}_2$ is given.

Firm 1’s problem induces a best response mapping

$$\vec{q}_1 = B(\vec{q}_2)$$

(Here $B$ maps a sequence into a sequence)

The Stackelberg leader’s problem is

$$\max_{\vec{q}_2} \Pi_2(B(\vec{q}_2), \vec{q}_2)$$
whose maximizer is a sequence $\tilde{q}_2$ that depends on the initial conditions $q_{10}, q_{20}$ and the parameters of the model $a_0, a_1, \gamma$.

This formulation captures key features of the model

- Both firms make once-and-for-all choices at time 0.
- This is true even though both firms are choosing sequences of quantities that are indexed by time.
- The Stackelberg leader chooses first within time 0, knowing that the Stackelberg follower will choose second within time 0.

While our abstract formulation reveals the timing protocol and equilibrium concept well, it obscures details that must be addressed when we want to compute and interpret a Stackelberg plan and the follower’s best response to it.

To gain insights about these things, we study them in more detail.

### 60.2.3 Firms’ Problems

Firm 1 acts as if firm 2’s sequence $\{q_{2t+1}\}_{t=0}^\infty$ is given and beyond its control.

Firm 2 knows that firm 1 chooses second and takes this into account in choosing $\{q_{2t+1}\}_{t=0}^\infty$.

In the spirit of working backwards, we study firm 1’s problem first, taking $\{q_{2t+1}\}_{t=0}^\infty$ as given.

We can formulate firm 1’s optimum problem in terms of the Lagrangian

$$L = \sum_{t=0}^\infty \beta^t \{a_0q_{1t} - a_1q_{1t}^2 - a_1q_{1t}q_{2t} - \gamma q_{1t}v_{1t}^2 + \lambda_t[q_{1t} + v_{1t} - q_{1t+1}]\}$$

Firm 1 seeks a maximum with respect to $\{q_{1t+1}, v_{1t}\}_{t=0}^\infty$ and a minimum with respect to $\{\lambda_t\}_{t=0}^\infty$.

We approach this problem using methods described in Ljungqvist and Sargent RMT5 chapter 2, appendix A and Macroeconomic Theory, 2nd edition, chapter IX.

First-order conditions for this problem are

$$\frac{\partial L}{\partial q_{1t}} = a_0 - 2a_1q_{1t} - a_1q_{2t} + \lambda_t - \beta^{-1}\lambda_{t-1} = 0, \quad t \geq 1$$

$$\frac{\partial L}{\partial v_{1t}} = -2\gamma q_{1t}v_{1t} + \lambda_t = 0, \quad t \geq 0$$

These first-order conditions and the constraint $q_{1t+1} = q_{1t} + v_{1t}$ can be rearranged to take the form

$$v_{1t} = \beta v_{1t+1} + \frac{\beta a_0}{2\gamma} - \frac{\beta a_1}{\gamma} q_{1t+1} - \frac{\beta a_1}{2\gamma} q_{2t+1}$$

$$q_{t+1} = q_{1t} + v_{1t}$$

We can substitute the second equation into the first equation to obtain

$$(q_{1t+1} - q_{1t}) = \beta(q_{1t+2} - q_{1t+1}) + c_0 - c_1q_{1t+1} - c_2q_{2t+1}$$
where \( c_0 = \frac{\beta a_0}{\gamma} \), \( c_1 = \frac{\beta a_1}{\gamma} \), \( c_2 = \frac{\beta a_1}{2\gamma} \).

This equation can in turn be rearranged to become the second-order difference equation

\[
q_{1t} + (1 + \beta + c_1)q_{1t+1} - \beta q_{1t+2} = c_0 - c_2 q_{2t+1}
\]  

Equation (1) is a second-order difference equation in the sequence \( \tilde{q}_1 \) whose solution we want.

It satisfies two boundary conditions:
- an initial condition that \( q_{1,0} \), which is given
- a terminal condition requiring that \( \lim_{T \to +\infty} \beta^T q_{1t}^2 < +\infty \)

Using the lag operators described in chapter IX of *Macroeconomic Theory, Second edition (1987)*, difference equation (1) can be written as

\[
\beta (1 - \frac{1 + \beta + c_1}{\beta} L + \beta^{-1} L^2) q_{1t+2} = -c_0 + c_2 q_{2t+1}
\]

The polynomial in the lag operator on the left side can be factored as

\[
(1 - \frac{1 + \beta + c_1}{\beta} L + \beta^{-1} L^2) = (1 - \delta_1 L)(1 - \delta_2 L)
\]

where \( 0 < \delta_1 < 1 < \frac{1}{\sqrt{\beta}} < \delta_2 \).

Because \( \delta_2 > \frac{1}{\sqrt{\beta}} \) the operator \( (1 - \delta_2 L) \) contributes an unstable component if solved backwards but a stable component if solved forwards.

Mechanically, write

\[
(1 - \delta_2 L) = -\delta_2 L (1 - \delta_2^{-1} L^{-1})
\]

and compute the following inverse operator

\[
[-\delta_2 L (1 - \delta_2^{-1} L^{-1})]^{-1} = -\delta_2 (1 - \delta_2^{-1})^{-1} L^{-1}
\]

Operating on both sides of equation (2) with \( \beta^{-1} \) times this inverse operator gives the follower’s decision rule for setting \( q_{1t+1} \) in the feedback-feedforward form

\[
q_{1t+1} = \delta_1 q_{1t} - c_0 \delta_2^{-1} \beta^{-1} \frac{1}{1 - \delta_2^{-1}} + c_2 \delta_2^{-1} \beta^{-1} \sum_{j=0}^{\infty} \delta_2^j q_{2t+j+1}, \quad t \geq 0
\]  

The problem of the Stackelberg leader firm 2 is to choose the sequence \( \{q_{2t+1}\}_{t=0}^{\infty} \) to maximize its discounted profits

\[
\sum_{t=0}^{\infty} \beta^t \left\{(a_0 - a_1 q_{1t} + q_{2t})q_{2t} - \gamma (q_{2t+1} - q_{2t})^2\right\}
\]

subject to the sequence of constraints (3) for \( t \geq 0 \).

We can put a sequence \( \{\theta_t\}_{t=0}^{\infty} \) of Lagrange multipliers on the sequence of equations (3) and formulate the following Lagrangian for the Stackelberg leader firm 2’s problem
\[ \bar{L} = \sum_{t=0}^{\infty} \beta^t \left\{ (a_0 - a_1(q_{1t} + q_{2t}))q_{2t} - \gamma(q_{2t+1} - q_{2t})^2 \right\} \\
+ \sum_{t=0}^{\infty} \beta^t \theta_t \left\{ \delta_1 q_{1t} - c_0 \delta_2^{-1} \beta^{-1} \frac{1}{1 - \delta_2} q_{1t} + c_2 \delta_2^{-1} \beta^{-1} \sum_{j=0}^{\infty} \delta_2^{-j} q_{2t+j+1} - q_{1t+1} \right\} \]  

(4)

subject to initial conditions for \( q_{1t}, q_{2t} \) at \( t = 0 \).

**Comments:** We have formulated the Stackelberg problem in a space of sequences.

The max-min problem associated with Lagrangian (4) is unpleasant because the time \( t \) component of firm 1’s payoff function depends on the entire future of its choices of \( \{q_{1t+j}\}_{j=0}^{\infty} \). This renders a direct attack on the problem cumbersome.

Therefore, below, we will formulate the Stackelberg leader’s problem recursively.

We’ll put our little duopoly model into a broader class of models with the same conceptual structure.

### 60.3 The Stackelberg Problem

We formulate a class of linear-quadratic Stackelberg leader-follower problems of which our duopoly model is an instance.

We use the optimal linear regulator (a.k.a. the linear-quadratic dynamic programming problem described in LQ Dynamic Programming problems) to represent a Stackelberg leader’s problem recursively.

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 \( t \).

In our duopoly example \( x_t = v_{1t} \), the time \( t \) decision of the Stackelberg follower.

Let \( u_t \) be a vector of decisions chosen by the Stackelberg leader at \( t \).

The \( z_t \) vector is inherited physically from the past.

But \( x_t \) is a decision made by the Stackelberg follower at time \( t \) that is the follower’s best response to the choice of an entire sequence of decisions made by the Stackelberg leader at time \( t = 0 \).

Let

\[ y_t = \begin{bmatrix} z_t \\ x_t \end{bmatrix} \]

Represent the Stackelberg leader’s one-period loss function as

\[ r(y, u) = y' R y + u' Q u \]

Subject to an initial condition for \( z_0 \), but not for \( x_0 \), the Stackelberg leader wants to maximize
\begin{equation}
- \sum_{t=0}^{\infty} \beta^t r(y_t, u_t)
\end{equation}

The Stackelberg leader faces the model

\begin{equation}
\begin{bmatrix}
I & 0 \\
G_{21} & G_{22}
\end{bmatrix}
\begin{bmatrix}
z_{t+1} \\
x_{t+1}
\end{bmatrix} =
\begin{bmatrix}
\hat{A}_{11} & \hat{A}_{12} \\
\hat{A}_{21} & \hat{A}_{22}
\end{bmatrix}
\begin{bmatrix}
z_t \\
x_t
\end{bmatrix} + \hat{B}u_t
\end{equation}

We assume that the matrix
\[
\begin{bmatrix}
I & 0 \\
G_{21} & G_{22}
\end{bmatrix}
\]
on the left side of equation (6) is invertible, so that we can multiply both sides by its inverse to obtain

\begin{equation}
\begin{bmatrix}
z_{t+1} \\
x_{t+1}
\end{bmatrix} =
\begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix}
\begin{bmatrix}
z_t \\
x_t
\end{bmatrix} + Bu_t
\end{equation}

or

\begin{equation}
y_{t+1} = Ay_t + Bu_t
\end{equation}

60.3.1 Interpretation of the Second Block of Equations

The Stackelberg follower’s best response mapping is summarized by the second block of equations of (7).

In particular, these equations are the first-order conditions of the Stackelberg follower’s optimization problem (i.e., its Euler equations).

These Euler equations summarize the forward-looking aspect of the follower’s behavior and express how its time \( t \) decision depends on the leader’s actions at times \( s \geq t \).

When combined with a stability condition to be imposed below, the Euler equations summarize the follower’s best response to the sequence of actions by the leader.

The Stackelberg leader maximizes (5) by choosing sequences \( \{u_t, x_t, z_{t+1}\}_{t=0}^{\infty} \) subject to (8) 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 Stackelberg leader uses its understanding of the responses restricted by (8) to manipulate the follower’s decisions.

60.3.2 More Mechanical Details

For any vector \( a_t \), define \( \bar{a}_t = [a_t, a_{t+1} \ldots] \).

Define a feasible set of \( (\bar{y}_1, \bar{u}_0) \) sequences

\[
\Omega(y_0) = \{(\bar{y}_1, \bar{u}_0) : y_{t+1} = Ay_t + Bu_t, \forall t \geq 0\}
\]

Please remember that the follower’s Euler equation is embedded in the system of dynamic equations \( y_{t+1} = Ay_t + Bu_t \).
Note that in the definition of $\Omega(y_0)$, $y_0$ is taken as given.

Although it is taken as given in $\Omega(y_0)$, eventually, the $x_0$ component of $y_0$ will be chosen by the Stackelberg leader.

### 60.3.3 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 $t \geq 0$.

Subproblem 2 is solved the **Stackelberg leader** at $t = 0$.

The two subproblems are designed

- to respect the protocol in which the follower chooses $\vec{q}_1$ after seeing $\vec{q}_2$ chosen by the leader
- to make the leader choose $\vec{q}_2$ while respecting that $\vec{q}_1$ will be the follower’s best response to $\vec{q}_2$
- to represent the leader’s problem recursively by artfully choosing the state variables confronting and the control variables available to the leader

#### Subproblem 1

$$v(y_0) = \max_{(\vec{y}_1, \vec{u}_0) \in \Omega(y_0)} \left\{ -r(y_t, u_t) + \beta \sum_{t=0}^{\infty} \beta^t r(y_t, u_t) \right\}$$

#### Subproblem 2

$$w(z_0) = \max_{x_0} v(y_0)$$

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, $z_0$.

### 60.3.4 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

$$v(y) = \max_{u, y^*} \left\{ -r(y, u) + \beta v(y^*) \right\}$$  \hspace{1cm} (9)

where the maximization is subject to
\[ y^* = Ay + Bu \]

and \( y^* \) denotes next period’s value.

Substituting \( v(y) = -y'Py \) into Bellman equation (9) gives

\[-y'Py = \max_{u,y^*} \{-y'Ry - u'Qu - \beta y'^*Py^*\}\]

which as in lecture linear regulator gives rise to the algebraic matrix Riccati equation

\[ P = R + \beta A'PA - \beta^2 A'PB(Q + \beta B'PB)^{-1}B'PA \]

and the optimal decision rule coefficient vector

\[ F = \beta(Q + \beta B'PB)^{-1}B'PA \]

where the optimal decision rule is

\[ u_t = -Fy_t \]

**Subproblem 2**

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

\[ x_0 = -P_{22}^{-1}P_{21}z_0 \]

### 60.4 Stackelberg Plan

Now let’s map our duopoly model into the above setup.

We’ll formulate a state space system

\[ y_t = \begin{bmatrix} z_t \\ x_t \end{bmatrix} \]

where in this instance \( x_t = v_{1t} \), the time \( t \) decision of the follower firm 1.

#### 60.4.1 Calculations to Prepare Duopoly Model

Now we’ll proceed to cast our duopoly model within the framework of the more general linear-quadratic structure described above.
That will allow us to compute a Stackelberg plan simply by enlisting a Riccati equation to solve a linear-quadratic dynamic program.

As emphasized above, firm 1 acts as if firm 2’s decisions \( \{q_{2t+1}, v_{2t}\}_{t=0}^{\infty} \) are given and beyond its control.

### 60.4.2 Firm 1’s Problem

We again formulate firm 1’s optimum problem in terms of the Lagrangian

\[
L = \sum_{t=0}^{\infty} \beta^t \{ a_0 q_{1t} - a_1 q_{1t}^2 - a_1 q_{1t} q_{2t} - \gamma v_{1t}^2 + \lambda_t [q_{1t} + v_{1t} - q_{1t+1}] \}
\]

Firm 1 seeks a maximum with respect to \( \{q_{1t+1}, v_{1t}\}_{t=0}^{\infty} \) and a minimum with respect to \( \{\lambda_t\}_{t=0}^{\infty} \).

First-order conditions for this problem are

\[
\frac{\partial L}{\partial q_{1t}} = a_0 - 2a_1 q_{1t} - a_1 q_{2t} + \lambda_t - \beta^{-1} \lambda_{t-1} = 0, \quad t \geq 1
\]

\[
\frac{\partial L}{\partial v_{1t}} = -2\gamma v_{1t} + \lambda_t = 0, \quad t \geq 0
\]

These first-order conditions and the constraint \( q_{1t+1} = q_{1t} + v_{1t} \) can be rearranged to take the form

\[
v_{1t} = \beta v_{1t+1} + \frac{\beta a_0}{2\gamma} - \frac{\beta a_1}{\gamma} q_{1t+1} - \frac{\beta a_1}{2\gamma} q_{2t+1}
\]

\[
q_{t+1} = q_{1t} + v_{1t}
\]

We use these two equations as components of the following linear system that confronts a Stackelberg continuation leader at time \( t \)

\[
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
\beta a_0 - \frac{\beta a_1}{2\gamma} - \alpha a_t \beta
\end{bmatrix}
\begin{bmatrix}
1 \\
q_{2t+1} \\
q_{1t+1} \\
v_{1t+1}
\end{bmatrix}
= \begin{bmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0
\end{bmatrix}
\begin{bmatrix}
1 \\
q_{2t} \\
q_{1t} \\
v_{1t}
\end{bmatrix}
+ \begin{bmatrix}
0 \\
0 \\
0 \\
1 \gamma v_{2t}
\end{bmatrix}
\]

Time \( t \) revenues of firm 2 are \( \pi_{2t} = a_0 q_{2t} - a_1 q_{2t}^2 - a_1 q_{1t} q_{2t} \) which evidently equal

\[
z_t^t R_t z_t = \begin{bmatrix}
q_{2t} \\
q_{1t}
\end{bmatrix}^t
\begin{bmatrix}
a_0 & 0 & \frac{a_0}{2} & 0 & 0 \\
0 & \frac{a_0}{2} & \frac{a_1}{2} & \frac{a_1}{2} & 0 \\
0 & 0 & \frac{a_1}{2} & \frac{a_1}{2} & 0
\end{bmatrix}
\begin{bmatrix}
q_{2t} \\
q_{1t}
\end{bmatrix}
\]

If we set \( Q = \gamma \), then firm 2’s period \( t \) profits can then be written

\[
y_t R y_t - Q v_{2t}^2
\]

where
CHAPTER 60. DYNAMIC STACKELBERG PROBLEMS

$$y_t = \begin{bmatrix} z_t \\ x_t \end{bmatrix}$$

with $x_t = v_{1t}$ and

$$R = \begin{bmatrix} R_1 & 0 \\ 0 & 0 \end{bmatrix}$$

We’ll report results of implementing this code soon.

But first we want to represent the Stackelberg leader’s optimal choices recursively.

It is important to do this for several reasons:

- properly to interpret a representation of the Stackelberg leader’s choice as a sequence of history-dependent functions
- to formulate a recursive version of the follower’s choice problem

First let’s get a recursive representation of the Stackelberg leader’s choice of $\tilde{q}_2$ for our duopoly model.

60.5 Recursive Representation of Stackelberg Plan

In order to attain an appropriate representation of the Stackelberg leader’s history-dependent plan, we will employ what amounts to a version of the Big K, little k device often used in macroeconomics by distinguishing $z_t$, which depends partly on decisions $x_t$ of the followers, from another vector $\tilde{z}_t$, which does not.

We will use $\tilde{z}_t$ and its history $\tilde{z}^t = [\tilde{z}_t, \tilde{z}_{t-1}, ..., \tilde{z}_0]$ to describe the sequence of the Stackelberg leader’s decisions that the Stackelberg follower takes as given.

Thus, we let $\tilde{y}_t = [\tilde{z}_t' \tilde{x}_t']$ with initial condition $\tilde{z}_0 = z_0$ given.

That we distinguish $\tilde{z}_t$ from $z_t$ is part and parcel of the Big K, little k device in this instance.

We have demonstrated that a Stackelberg plan for $\{u_t\}_{t=0}^\infty$ has a recursive representation

$$\tilde{x}_0 = -P_{22}^{-1} P_{21} z_0$$
$$u_t = -F \tilde{y}_t, \quad t \geq 0$$
$$\tilde{y}_{t+1} = (A - BF) \tilde{y}_t, \quad t \geq 0$$

From this representation we can deduce the sequence of functions $\sigma = \{\sigma_t(\tilde{z}^t)\}_{t=0}^\infty$ that comprise a Stackelberg plan.

For convenience, let $\tilde{A} \equiv A - BF$ and partition $\tilde{A}$ conformably to the partition $y_t = \begin{bmatrix} \tilde{z}_t \\ \tilde{x}_t \end{bmatrix}$ as

$$\begin{bmatrix} \tilde{A}_{11} & \tilde{A}_{12} \\ \tilde{A}_{21} & \tilde{A}_{22} \end{bmatrix}$$

Let $H_0^0 \equiv -P_{22}^{-1} P_{21}$ so that $\tilde{x}_0 = H_0^0 \tilde{z}_0$. 
Then iterations on \( \dot{y}_{t+1} = A \dot{y}_t \) starting from initial condition \( \dot{y}_0 = \begin{bmatrix} \dot{z}_0 \\ H_0^0 z_0 \end{bmatrix} \) imply that for \( t \geq 1 \)

\[
x_t = \sum_{j=1}^{t} H^t_j \ddot{z}_{t-j}
\]

where

\[
H^1_1 = \ddot{A}_{21} \\
H^2_1 = \ddot{A}_{22} \ddot{A}_{21} \\
\vdots \\
H^{t-1}_1 = \ddot{A}^{t-2} \ddot{A}_{21} \\
H^t_1 = \ddot{A}^{t-1} (\ddot{A}_{21} + \ddot{A}_{22} H^0_0)
\]

An optimal decision rule for the Stackelberg’s choice of \( u_t \) is

\[
u_t = -F \ddot{y}_t = - \begin{bmatrix} F_z & F_x \end{bmatrix} \begin{bmatrix} \ddot{z}_t \\ x_t \end{bmatrix}
\]

or

\[
u_t = -F_z \ddot{z}_t - F_x \sum_{j=1}^{t} H^t_j \ddot{z}_{t-j} = \sigma_t(\ddot{z}^t)
\]

Representation (10) confirms that whenever \( F_x \neq 0 \), the typical situation, the time \( t \) component \( \sigma_t \) of a Stackelberg plan is **history dependent**, meaning that the Stackelberg leader’s choice \( u_t \) depends not just on \( \ddot{z}_t \) but on components of \( \ddot{z}^{t-1} \).

### 60.5.1 Comments and Interpretations

After all, at the end of the day, it will turn out that because we set \( \ddot{z}_0 = z_0 \), it will be true that \( z_t = \ddot{z}_t \) for all \( t \geq 0 \).

Then why did we distinguish \( \ddot{z}_t \) from \( z_t \)?

The answer is that if we want to present to the Stackelberg **follower** a history-dependent representation of the Stackelberg **leader’s** sequence \( \ddot{q}_2 \), we must use representation (10) cast in terms of the history \( \ddot{z}^t \) and **not** a corresponding representation cast in terms of \( z^t \).

### 60.5.2 Dynamic Programming and Time Consistency of follower’s Problem

Given the sequence \( \ddot{q}_2 \) chosen by the Stackelberg leader in our duopoly model, it turns out that the Stackelberg **follower’s** problem is recursive in the **natural** state variables that confront a follower at any time \( t \geq 0 \).

This means that the follower’s plan is time consistent.
To verify these claims, we’ll formulate a recursive version of a follower’s problem that builds on our recursive representation of the Stackelberg leader’s plan and our use of the Big K, little k idea.

60.5.3 Recursive Formulation of a Follower’s Problem

We now use what amounts to another “Big K, little k” trick (see rational expectations equilibrium) to formulate a recursive version of a follower’s problem cast in terms of an ordinary Bellman equation.

Firm 1, the follower, faces $\{q_{2t}\}_{t=0}^{\infty}$ as a given quantity sequence chosen by the leader and believes that its output price at $t$ satisfies

$$p_t = a_0 - a_1(q_{1t} + q_{2t}), \quad t \geq 0$$

Our challenge is to represent $\{q_{2t}\}_{t=0}^{\infty}$ as a given sequence.

To do so, recall that under the Stackelberg plan, firm 2 sets output according to the $q_{2t}$ component of

$$y_{t+1} = \begin{bmatrix} 1 \\ q_{2t} \\ q_{1t} \\ x_t \end{bmatrix}$$

which is governed by

$$y_{t+1} = (A - BF)y_t$$

To obtain a recursive representation of a $\{q_{2t}\}$ sequence that is exogenous to firm 1, we define a state $\tilde{y}_t$

$$\tilde{y}_t = \begin{bmatrix} 1 \\ \tilde{q}_{2t} \\ \tilde{q}_{1t} \\ \tilde{x}_t \end{bmatrix}$$

that evolves according to

$$\tilde{y}_{t+1} = (A - BF)\tilde{y}_t$$

subject to the initial condition $\tilde{q}_{10} = q_{10}$ and $\tilde{x}_0 = x_0$ where $x_0 = -P_{22}^{-1}P_{21}$ as stated above.

Firm 1’s state vector is

$$X_t = \begin{bmatrix} \tilde{y}_t \\ q_{1t} \end{bmatrix}$$

It follows that the follower firm 1 faces law of motion
60.5. RECURSIVE REPRESENTATION OF STACKELBERG PLAN

\[
\begin{bmatrix}
\tilde{y}_{t+1} \\
q_{t+1}
\end{bmatrix} = \begin{bmatrix}
A - BF & 0 \\
0 & 1
\end{bmatrix} \begin{bmatrix}
\tilde{y}_t \\
q_{1t}
\end{bmatrix} + \begin{bmatrix}
0 \\
1
\end{bmatrix} x_t
\]  

(11)

This specification assures that from the point of view of a firm 1, \(q_{2t}\) is an exogenous process.

Here

- \(\tilde{q}_{1t}, \tilde{x}_t\) play the role of \textbf{Big K}.
- \(q_{1t}, x_t\) play the role of \textbf{little k}.

The time \(t\) component of firm 1’s objective is

\[
\tilde{X}^t \tilde{R} x_t - x_t^2 \tilde{Q} = \begin{bmatrix}
1 \\
q_{2t} \\
\tilde{q}_{1t} \\
\tilde{x}_t \\
q_{1t}
\end{bmatrix}' \begin{bmatrix}
0 & 0 & 0 & 0 & \frac{a_2}{2} \\
0 & 0 & 0 & 0 & -\frac{a_2}{2} \\
0 & 0 & 0 & 0 & 0 \\
\frac{a_0}{2} & 0 & 0 & 0 & -a_1 \\
0 & 0 & 0 & 0 & 0
\end{bmatrix} \begin{bmatrix}
1 \\
q_{2t} \\
\tilde{q}_{1t} \\
\tilde{x}_t \\
q_{1t}
\end{bmatrix} - \gamma x_t^2
\]

Firm 1’s optimal decision rule is

\[x_t = -\tilde{F} X_t\]

and it’s state evolves according to

\[
\tilde{X}_{t+1} = (\tilde{A} - \tilde{B} \tilde{F}) X_t
\]

under its optimal decision rule.

Later we shall compute \(\tilde{F}\) and verify that when we set

\[
X_0 = \begin{bmatrix}
1 \\
q_{20} \\
q_{10} \\
x_0 \\
q_{10}
\end{bmatrix}
\]

we recover

\[x_0 = -\tilde{F} \tilde{X}_0\]

which will verify that we have properly set up a recursive representation of the follower’s problem facing the Stackelberg leader’s \(\tilde{q}_2\).

60.5.4 Time Consistency of Follower’s Plan

Since the follower can solve its problem using dynamic programming its problem is recursive in what for it are the \textbf{natural state variables}, namely
It follows that the follower’s plan is time consistent.

### 60.6 Computing the Stackelberg Plan

Here is our code to compute a Stackelberg plan via a linear-quadratic dynamic program as outlined above

In [1]: using InstantiateFromURL
       # optionally add arguments to force installation: instantiate = true,
       # precompile = true
       github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

In [2]: using QuantEcon, Plots, LinearAlgebra, Statistics, Parameters, Random
       gr(fmt = :png);

We define named tuples and default values for the model and solver settings, and instantiate one copy of each

In [3]: model = @with_kw (a0 = 10, 
                        a1 = 2, 
                        β = 0.96, 
                        γ = 120., 
                        n = 300)

settings = @with_kw (tol0 = 1e-8, 
                      tol1 = 1e-16, 
                      tol2 = 1e-2)

defaultModel = model();
defaultSettings = settings();

Now we can compute the actual policy using the LQ routine from QuantEcon.jl

In [4]: @unpack a0, a1, β, γ, n = defaultModel
       @unpack tol0, tol1, tol2 = defaultSettings

βs = [β^x for x = 0:n-1]
Alhs = I + zeros(4, 4);
Alhs[4, :] = [β * a0 / (2 * γ), -β * a1 / (2 * γ), -β * a1 / γ, β] # Euler equation coefficients
Arhs = I + zeros(4, 4);
Arhs[3, 4] = 1.;
Alhsinv = inv(Alhs);
60.6. COMPUTING THE STACKELBERG PLAN

\[ A = \text{Alhsinv} \times \text{Arhs}; \]
\[ B = \text{Alhsinv} \times [0, 1, 0, 0, \ldots]; \]
\[ R = [0 -a0/2 0 0; -a0/2 a1 a1/2 0; 0 a1/2 0 0; 0 0 0 0]; \]
\[ Q = \gamma; \]
\[ \text{lq} = \text{QuantEcon.LQ}(Q, R, A, B, \text{bet} = \beta); \]
\[ P, F, d = \text{stationary_values(lq)} \]
\[ P22 = P[4:end, 4:end]; \]
\[ P21 = P[4:end, 1:3]; \]
\[ P22inv = \text{inv}(P22); \]
\[ H_0_0 = -P22inv \times P21; \]

```plaintext
# simulate forward
π_leader = zeros(n);
z0 = [1, 1, 1];
x0 = H_0_0 \times z0;
y0 = vcat(z0, x0);
Random.seed!(1) # for reproducibility
yt, ut = compute_sequence(lq, y0, n);
π_matrix = R + F' \times Q \times F;

for t in 1:n
    π_leader[t] = -(yt[:, t]' \times π_matrix \times yt[:, t]);
end
println("Computed policy for Stackelberg leader: \$F")
```

Computed policy for Stackelberg leader: [-1.5800445387726552 0.294613127470314 0.6748093760774969 6.539705936147513]

60.6.1 Implied Time Series for Price and Quantities

The following code plots the price and quantities

```
In [5]: q_leader = yt[2, 1:end];
q_follower = yt[3, 1:end];
q = q_leader + q_follower;
p = a0 .- a1*q;
plot(1:n+1, [q_leader, q_follower, p], title = "Output and Prices, Stackelberg Duopoly", labels = ["leader output", "follower output", "price"], xlabel = "t")
```

Out[5]:
60.6.2 Value of Stackelberg Leader

We’ll compute the present value earned by the Stackelberg leader.
We’ll compute it two ways (they give identical answers – just a check on coding and thinking)

In [6]:

```python
v_leader_forward = sum(βs * π_leader);
v_leader_direct = -yt[:, 1]' * P * yt[:, 1];
println("v_leader_forward (forward sim) is $v_leader_forward")
println("v_leader_direct is $v_leader_direct")
```

```
v_leader_forward (forward sim) is 150.0316212532548
v_leader_direct is 150.03237147548847
```

In [7]:

```python
# manually check whether P is an approximate fixed point
P_next = (R + F' * Q * F + β * (A - B * F)' * P * (A - B * F));
all(P - P_next .< tol0)
```

```
Out[7]: true
```

In [8]:

```python
# manually checks whether two different ways of computing the
# value function give approximately the same answer
v_expanded = -((y0' * R * yθ + ut[:, 1]' * Q * ut[:, 1] +
       β * (y0' * (A - B * F)' * P * (A - B * F) * yθ)));
(v_leader_direct - v_expanded < tol0)[1, 1]
```

```
Out[8]: true
```
60.7 Exhibiting Time Inconsistency of Stackelberg Plan

In the code below we compare two values
- the continuation value \(-y_t P y_t\) earned by a continuation Stackelberg leader who inherits state \(y_t\) at \(t\)
- the value of a reborn Stackelberg leader who inherits state \(z_t\) at \(t\) and sets \(x_t = -P_{21}^{-1} P_{22}\)

The difference between these two values is a tell-tale time of the time inconsistency of the Stackelberg plan

In [9]: # Compute value function over time with reset at time t
   vt_leader = zeros(n);
   vt_reset_leader = similar(vt_leader);
   yt_reset = copy(yt); 
   yt_reset[end, :] = (H_0_0 * yt[1:3, :])
   for t in 1:n
       vt_leader[t] = -yt[:, t]' * P * yt[:, t]
       vt_reset_leader[t] = -yt_reset[:, t]' * P * yt_reset[:, t]
   end
   p1 = plot(1:n+1, [(-F * yt)', (-F * yt_reset)'], labels = ["Stackelberg\n   "Leader", "Continuation Leader at t"],
         title = "Leader Control Variable", xlabel = "t");
   p2 = plot(1:n+1, [yt[4, :], yt_reset[4, :]], title = "Follower Control\n   Variable", xlabel = "t", legend = false);
   p3 = plot(1:n, [vt_leader, vt_reset_leader], legend = false,
         xlabel = "t", title = "Leader Value Function");
   plot(p1, p2, p3, layout = (3, 1), size = (800, 600))

Out[9]:
60.8 Recursive Formulation of the Follower’s Problem

We now formulate and compute the recursive version of the follower’s problem.

We check that the recursive Big $K$, little $k$ formulation of the follower’s problem produces
the same output path $\hat{q}_1$ that we computed when we solved the Stackelberg problem

In [10]:
$\bar{A} = I + \text{zeros}(5, 5)$;
$\bar{A}[1:4, 1:4] = A - B \times F$;
$R = [0 0 0 -a0/2; 0 0 0 a1/2; 0 0 0 0; 0 0 0 0; -a0/2 a1/2 0 0 a1]$;
$Q = Q$;
$B = [0, 0, 0, 1]$;

$lq_{\text{tilde}} = \text{QuantEcon.LQ}(Q, R, \bar{A}, B, \text{bet}=\beta)$;
$p, \bar{F}, d = \text{stationary_values}(lq_{\text{tilde}});
\hat{y}_0_{\text{tilde}} = vcat(y0, y0[3]);
\hat{y}t_{\text{tilde}} = \text{compute_sequence}(lq_{\text{tilde}}, \hat{y}_0_{\text{tilde}}, n)[1]$;

In [11]:
# checks that the recursive formulation of the follower's problem gives
# the same solution as the original Stackelberg problem
plot(1:n+1, [yt_tilde[5, :], yt_tilde[3, :]], labels = ["q_tilde", "q"])
Note: Variables with \_tilde are obtained from solving the follower’s problem – those without are from the Stackelberg problem.

\begin{verbatim}
In [12]: # maximum absolute difference in quantities over time between the first and second solution methods
    max(abs(yt_tilde[5] - yt_tilde[3]))
Out[12]: 0.0

In [13]: # x0 == x0_tilde
    yt[:, 1][end] - (yt_tilde[:, 2] - yt_tilde[:, 1])[end] < tol0
Out[13]: True
\end{verbatim}

\textbf{60.8.1 Explanation of Alignment}

If we inspect the coefficients in the decision rule \( \tilde{F} \), we can spot the reason that the follower chooses to set \( x_t = \tilde{x}_t \) when it sets \( x_t = -\tilde{F}X_t \) in the recursive formulation of the follower problem.

Can you spot what features of \( \tilde{F} \) imply this?

Hint: remember the components of \( X_t \)

\begin{verbatim}
In [14]: \tilde{F} # policy function in the follower's problem
Out[14]: 1x5 Array{Float64,2}:
    2.5489e-17  -3.18612e-18  -0.103187  -1.0  0.103187
\end{verbatim}
In [15]: P # value function in the Stackelberg problem

Out[15]: 4×4 Array{Float64,2}:
    963.541  -194.605  -511.622  -5258.23
   -194.605   37.3536   81.9771   784.765
  -511.622   81.9771  247.343  2517.05
  -5258.23  784.765  2517.05  25556.2

In [16]: P̃ # value function in the follower's problem

Out[16]: 5×5 Array{Float64,2}:
       -18.1991   2.58003  15.6049   151.23  -5.0
       2.58003  -0.969466  -5.26008  -50.9764  1.0
      15.6049  -5.26008  -32.2759  -312.792 -12.3824
     151.23  -50.9764  -312.792  -3031.33 -120.0
      -5.0   1.0   -12.3824  -120.0   14.3824

In [17]: # manually check that P is an approximate fixed point
   all((P - ((R + F' * Q * F) + β * (A - B * F)' * P * (A - B * F)).< tol))

Out[17]: true

In [18]: # compute `P_guess` using `F_tilde_star`
   F̃_star = [0, 0, 0, 1, 0]';
   P_guess = zeros(5, 5);
   for i in 1:1000
       P_guess = ((R + F̃_star' * Q * F̃_star) +
                   β * (Ã - B̃ * F̃_star)' * P_guess
                   * (Ã - B̃ * F̃_star));
   end

In [19]: # value function in the follower's problem
   -(y0̃_tilde' * P̃ * y0̃_tilde)[1, 1]

Out[19]: 112.65590740578102

In [20]: # value function using P_guess
   -(y0̃_tilde' * P_guess * y0̃_tilde)[1, 1]

Out[20]: 112.65590740578097

In [21]: # c policy using policy iteration algorithm
   F_iter = (β * inv(Q̃ + B̃' * P_guess * B̃)
            * B̃' * P_guess * Ã);
   P_iter = zeros(5, 5);
   dist_vec = zeros(5, 5);
for i in 1:100
    # compute P_iter
    dist_vec = similar(P_iter)
    for j in 1:1000
        P_iter = \( (\bar{R} + F_iter' * Q * F_iter) + \beta * (\bar{A} - \bar{B} * F_iter)' * P_iter * (\bar{A} - \bar{B} * F_iter) \);
    # update F_iter
    F_iter = \( \beta * \text{inv}(Q + \beta * \bar{B}' * P_iter * \bar{B}) * \bar{B}' * P_iter * \bar{A} \);
    dist_vec = P_iter - \( (\bar{R} + F_iter' * Q * F_iter) + \beta * (\bar{A} - \bar{B} * F_iter)' * P_iter * (\bar{A} - \bar{B} * F_iter) \);
end
end
if maximum(abs.(dist_vec)) < 1e-8
    dist_vec2 = F_iter - \( (\beta * \text{inv}(Q + \beta * \bar{B}' * P_iter * \bar{B}) * \bar{B}' * P_iter * \bar{A}) \)
    if maximum(abs.(dist_vec2)) < 1e-8
       @show F_iter
    else
        println("The policy didn't converge: try increasing the number of outer loop iterations")
    end
else
    println("The policy didn't converge: try increasing the number of inner loop iterations")
end

F_iter = [0.0 -1.474514954580286e-17 -0.10318650145223823 -1.0 0.10318650145223823]

Out[21]: 1×5 Adjoint{Float64,Array{Float64,1}}:
0.0  -1.47451e-17  -0.103187  -1.0  0.103187

In [22]: yt_tilde_star = zeros(n, 5);
yt_tilde_star[1, :] = y0_tilde;
for t in 1:n-1
    yt_tilde_star[t+1, :] = (\bar{A} - \bar{B} * \bar{F}_star) * yt_tilde_star[t, :];
end
plot([yt_tilde_star[:, 5], yt_tilde[3, :]], labels = ["q_tilde", "q"])

Out[22]:
In [23]: \text{maximum}(\text{abs}(\text{yt\_tilde\_star}[; , 5] - \text{yt\_tilde}[3, 1:\text{end}\_1]))

Out[23]: 0.0

60.9 Markov Perfect Equilibrium

The state vector is

\[ z_t = \begin{bmatrix} 1 \\ q_{2t} \\ q_{1t} \end{bmatrix} \]

and the state transition dynamics are

\[ z_{t+1} = A z_t + B_1 v_{1t} + B_2 v_{2t} \]

where \( A \) is a 3 \times 3 identity matrix and

\[ B_1 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \quad B_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \]

The Markov perfect decision rules are

\[ v_{1t} = -F_1 z_t, \quad v_{2t} = -F_2 z_t \]

and in the Markov perfect equilibrium the state evolves according to
\[ z_{t+1} = (A - B_1 F_1 - B_2 F_2) z_t \]

In [24]: # in LQ form
A = I + zeros(3, 3);
B1 = [0, 0, 1];
B2 = [0, 1, 0];
R1 = [0 0 -a0/2; 0 0 a1/2; -a0/2 a1/2 a1];
R2 = [0 -a0/2 0; -a0/2 a1 a1/2; 0 a1/2 0];
Q1 = Q2 = γ;
S1 = S2 = W1 = W2 = M1 = M2 = 0.;

# solve using nnash from QE
F1, F2, P1, P2 = nnash(A, B1, B2, R1, R2, Q1, Q2, S1, S2, W1, W2, M1, M2, beta = β, tol = tol1);

# simulate forward
AF = A - B1 * F1 - B2 * F2;
z = zeros(3, n);
z[:, 1] .= 1;
for t in 1:n-1
    z[:, t+1] = AF * z[:, t]
end
println("Policy for F1 is \$F1\")
println("Policy for F2 is \$F2\")

Policy for F1 is [-0.22701362843207126 0.03129874118441059 0.09447112842804818]
Policy for F2 is [-0.22701362843207126 0.09447112842804818 0.03129874118441059]

In [25]: q1 = z[2, :];
q2 = z[3, :];
q = q1 + q2; # total output, MPE
p = a0 .- a1 * q; # total price, MPE
plot([q, p], labels = ["total output", "total price"], title = "Output and prices,”
    duopoly MPE", xlabel = "t")

Out[25]:
## CHAPTER 60. DYNAMIC STACKELBERG PROBLEMS

In [26]: # computes the maximum difference in quantities across firms
   \[
   \text{maximum}(\text{abs}(q_1 - q_2))
   \]

Out[26]: 8.881784197001252e-16

In [27]: # compute values
   \[
   u_1 = -F_1 \ast z;
   u_2 = -F_2 \ast z;
   \pi_1 = (p \ast q_1)' - \gamma \ast u_1.^2;
   \pi_2 = (p \ast q_2)' - \gamma \ast u_2.^2;
   \]

   \[
   v_{1\text{ forward}} = \pi_1 \ast \beta s;
   v_{2\text{ forward}} = \pi_2 \ast \beta s;
   \]

   v_{1\text{ direct}} = -z[:, 1]' \ast P_1 \ast z[:, 1];
   v_{2\text{ direct}} = -z[:, 1]' \ast P_2 \ast z[:, 1];

   \]

   println("Firm 1: Direct is \$(v_{1\text{ direct}}), Forward is \$(v_{1\text{ forward}[1])");
   println("Firm 2: Direct is \$(v_{2\text{ direct}}), Forward is \$(v_{2\text{ forward}[1])");

Firm 1: Direct is 133.329555721595, Forward is 133.33033197956638
Firm 2: Direct is 133.3295557215945, Forward is 133.33033197956638

In [28]: # sanity check
   \[
   \Lambda_1 = A - B_2 \ast F_2;
   lq_1 = \text{QuantEcon.LQ}(Q_1, R_1, \Lambda_1, B_1, \text{bet} = \beta);
   P_{1\_ih}, F_{1\_ih}, d = \text{stationary_values}(lq_1);
   \]

   v_{2\text{ direct alt}} = -z[:, 1]' \ast P_{1\_ih} \ast z[:, 1] + d;
   all(abs.(v_{2\text{ direct}} - v_{2\text{ direct alt}}) < \text{tol2})

Out[28]: true
60.10 MPE vs. Stackelberg

In [29]: vt_MPE = zeros(n);
   vt_follower = zeros(n);

   for t in 1:n
       vt_MPE[t] = -z[:, t]' * P1 * z[:, t];
       vt_follower[t] = -yt_tilde[:, t]' * P * yt_tilde[:, t];
   end

   plot([vt_MPE, vt_leader, vt_follower], labels = ["MPE", "Stackelberg leader", "Stackelberg follower"], title = "MPE vs Stackelberg Values", xlabel = "t")

Out[29]:

In [30]: # display values
   println("vt_leader(y0) = 
   println("vt_follower(y0) = 
   println("vt_MPE(y0) = 

   vt_leader(y0) = 150.03237147548847
   vt_follower(y0) = 112.65590740578102
   vt_MPE(y0) = 133.3295555721595

In [31]: # total difference in value b/t Stackelberg and MPE

Out[31]: -3.9708322630494877
Chapter 61

Optimal Taxation in an LQ Economy

61.1 Contents

- Overview 61.2
- The Ramsey Problem 61.3
- Implementation 61.4
- Examples 61.5
- Exercises 61.6
- Solutions 61.7

61.2 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 [72] 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 [88] 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
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 [8].

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.

61.2.1 Model Features

- Linear quadratic (LQ) model
- Representative household
- Stochastic dynamic programming over an infinite horizon
- Distortionary taxation

61.2.2 Setup

In [1]: using InstantiateFromURL
   
   # optionally add arguments to force installation: instantiate = true,
   
   ~precompile = true
   
   github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

In [2]: using LinearAlgebra, Statistics

61.3 The Ramsey Problem

We begin by outlining the key assumptions regarding technology, households and the government sector.

61.3.1 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.

61.3.2 Households

Consider a representative household who chooses a path \(\{\ell_t, c_t\}\) for labor and consumption to maximize

\[
-\mathbb{E}\frac{1}{2} \sum_{t=0}^{\infty} \beta^t \left[ (c_t - b_t)^2 + \ell_t^2 \right]
\]  

(1)
subject to the budget constraint

\[ \mathbb{E} \sum_{t=0}^{\infty} \beta^t p_t^0 [d_t + (1 - \tau_t)\ell_t + s_t - c_t] = 0 \]  \hspace{1cm} (2)

Here

- \( \beta \) is a discount factor in \( (0, 1) \)
- \( p_t^0 \) is a scaled Arrow-Debreu price at time 0 of history contingent goods at time \( t + j \)
- \( b_t \) is a stochastic preference parameter
- \( d_t \) is an endowment process
- \( \tau_t \) is a flat tax rate on labor income
- \( s_t \) is a promised time-\( t \) coupon payment on debt issued by the government

The scaled Arrow-Debreu price \( p_t^0 \) is related to the unscaled Arrow-Debreu price as follows.

If we let \( \pi_t^0(x^t) \) denote the probability (density) of a history \( x^t = [x_t, x_{t-1}, \ldots, x_0] \) of the state \( x^t \), then the Arrow-Debreu time 0 price of a claim on one unit of consumption at date \( t \), history \( x^t \) would be

\[ \frac{\beta^t p_t^0}{\pi_t^0(x^t)} \]

Thus, our scaled Arrow-Debreu price is the ordinary Arrow-Debreu price multiplied by the discount factor \( \beta^t \) and divided by an appropriate probability.

The budget constraint \( (2) \) requires that the present value of consumption be restricted to equal the present value of endowments, labor income and coupon payments on bond holdings.

### 61.3.3 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.

### 61.3.4 Exogenous Variables

Endowments, government expenditure, the preference shock process \( b_t \), and promised coupon payments on initial government debt \( s_t \) are all exogenous, and given by

- \( d_t = S_d x_t \)
- \( g_t = S_g x_t \)
- \( b_t = S_b x_t \)
- \( s_t = S_s x_t \)

The matrices \( S_d, S_g, S_b, S_s \) are primitives and \( \{x_t\} \) is an exogenous stochastic process taking values in \( \mathbb{R}^k \).
We consider two specifications for \( \{x_t\} \).

1. Discrete case: \( \{x_t\} \) is a discrete state Markov chain with transition matrix \( P \).

2. VAR case: \( \{x_t\} \) obeys \( x_{t+1} = Ax_t + Cw_{t+1} \) where \( \{w_t\} \) is independent zero mean Gaussian with identity covariance matrix.

### 61.3.5 Feasibility

The period-by-period feasibility restriction for this economy is

\[
c_t + g_t = d_t + \ell_t
\]  

(3)

A labor-consumption process \( \{\ell_t, c_t\} \) is called feasible if (3) holds for all \( t \).

### 61.3.6 Government budget constraint

Where \( p^0_t \) is again a scaled Arrow-Debreu price, the time zero government budget constraint is

\[
\mathbb{E} \sum_{t=0}^{\infty} \beta^t p^0_t (s_t + g_t - \tau_t \ell_t) = 0
\]  

(4)

### 61.3.7 Equilibrium

An equilibrium is a feasible allocation \( \{\ell_t, c_t\} \), a sequence of prices \( \{p^0_t\} \), and a tax system \( \{\tau_t\} \) such that

1. The allocation \( \{\ell_t, c_t\} \) is optimal for the household given \( \{p^0_t\} \) and \( \{\tau_t\} \).

2. The government’s budget constraint (4) is satisfied.

The Ramsey problem is to choose the equilibrium \( \{\ell_t, c_t, \tau_t, p^0_t\} \) that maximizes the household’s welfare.

If \( \{\ell_t, c_t, \tau_t, p^0_t\} \) solves the Ramsey problem, then \( \{\tau_t\} \) is called the Ramsey plan.

The solution procedure we adopt is

1. Use the first-order conditions from the household problem to pin down prices and allocations given \( \{\tau_t\} \).

2. Use these expressions to rewrite the government budget constraint (4) in terms of exogenous variables and allocations.

3. Maximize the household’s objective function (1) subject to the constraint constructed in step 2 and the feasibility constraint (3).

The solution to this maximization problem pins down all quantities of interest.
61.3. THE RAMSEY PROBLEM

61.3.8 Solution

Step one is to obtain the first-conditions for the household’s problem, taking taxes and prices as given.

Letting $\mu$ be the Lagrange multiplier on (2), 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

$$p_t^0 = \frac{b_t - c_t}{b_0 - c_0} \quad \text{and} \quad \tau_t = 1 - \frac{\ell_t}{b_t - c_t} \quad (5)$$

Substituting (5) into the government’s budget constraint (4) yields

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t \left[ (b_t - c_t)(s_t + g_t - \ell_t) + \ell_t^2 \right] = 0 \quad (6)$$

The Ramsey problem now amounts to maximizing (1) subject to (6) and (3).

The associated Lagrangian is

$$\mathcal{L} = \mathbb{E} \sum_{t=0}^{\infty} \beta^t \left\{ -\frac{1}{2} \left[(c_t - b_t)^2 + \ell_t^2 \right] + \lambda \left[ (b_t - c_t)(\ell_t - s_t - g_t) - \ell_t^2 \right] + \mu_t [d_t + \ell_t - c_t - g_t] \right\} \quad (7)$$

The first order conditions associated with $c_t$ and $\ell_t$ are

$$-(c_t - b_t) + \lambda[-\ell_t + (g_t + s_t)] = \mu_t$$

and

$$\ell_t - \lambda[(b_t - c_t) - 2\ell_t] = \mu_t$$

Combining these last two equalities with (3) and working through the algebra, one can show that

$$\ell_t = \tilde{\ell}_t - \nu m_t \quad \text{and} \quad c_t = \tilde{c}_t - \nu m_t \quad (8)$$

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 government’s budget constraint again.

The term inside the brackets in (6) is $(b_t - c_t)(s_t + g_t) - (b_t - c_t)\ell_t + \ell_t^2$.

Using (8), the definitions above and the fact that $\tilde{\ell} = b - \tilde{c}$, this term can be rewritten as
\[ (b_t - \bar{c}_t)(g_t + s_t) + 2m_t^2(\nu^2 - \nu) \]

Reinserting into (6), we get

\[
\mathbb{E}\left\{ \sum_{t=0}^{\infty} \beta^t (b_t - \bar{c}_t)(g_t + s_t) \right\} + (\nu^2 - \nu)\mathbb{E}\left\{ \sum_{t=0}^{\infty} \beta^t 2m_t^2 \right\} = 0 \quad (9)
\]

Although it might not be clear yet, we are nearly there because:

- The two expectations terms in (9) 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 (8).
- Once we have the allocations, prices and the tax system can be derived from (5).

### 61.3.9 Computing the Quadratic Term

Let’s consider how to obtain the term \( \nu \) in (9).

If we can compute the two expected geometric sums

\[
b_0 := \mathbb{E}\left\{ \sum_{t=0}^{\infty} \beta^t (b_t - \bar{c}_t)(g_t + s_t) \right\} \quad \text{and} \quad a_0 := \mathbb{E}\left\{ \sum_{t=0}^{\infty} \beta^t 2m_t^2 \right\} \quad (10)
\]

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 (10).

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 (10), the random variable \( 2m_t^2 \) can be written as

\[
\frac{1}{2} x'_t(S_b - S_d - S_s)(S_b - S_d + S_s)x_t
\]

It follows that both objects of interest are special cases of the expression

\[
q(x_0) = \mathbb{E}\sum_{t=0}^{\infty} \beta^t x'_t H x_t \quad (11)
\]

where \( H \) is a matrix conformable to \( x_t \) and \( x'_t \) is the transpose of column vector \( x_t \).
61.3. THE RAMSEY PROBLEM

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.

61.3.10 Finite state Markov case

Next suppose that \( \{x_t\} \) is the discrete Markov process described above.

Suppose further that each \( x_t \) takes values in the state space \( \{x^1, \ldots, x^N\} \subset \mathbb{R}^k \).

Let \( h: \mathbb{R}^k \to \mathbb{R} \) be a given function, and suppose that we wish to evaluate

\[
q(x_0) = \mathbb{E} \sum_{t=0}^{\infty} \beta^t h(x_t) \quad \text{given} \quad x_0 = x^j
\]

For example, in the discussion above, \( h(x_t) = x_t'Hx_t \).

It is legitimate to pass the expectation through the sum, leading to

\[
q(x_0) = \sum_{t=0}^{\infty} \beta^t (P^t h)[j] \quad \text{(12)}
\]

Here

- \( P^t \) is the \( t \)-th power of the transition matrix \( P \)
- \( h \) is, with some abuse of notation, the vector \( (h(x^1), \ldots, h(x^N)) \)
- \( (P^t h)[j] \) indicates the \( j \)-th element of \( P^t h \)

It can be show that (12) 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.

61.3.11 Other Variables

We are interested in tracking several other variables besides the ones described above.

To prepare the way for this, we define

\[
p^t_{t+j} = \frac{b_{t+j} - c_{t+j}}{b_t - c_t}
\]

as the scaled Arrow-Debreu time \( t \) price of a history contingent claim on one unit of consumption at time \( t + j \).

These are prices that would prevail at time \( t \) if market were reopened at time \( t \).

These prices are constituents of the present value of government obligations outstanding at time \( t \), which can be expressed as
Using our expression for prices and the Ramsey plan, we can also write $B_t$ as

$$B_t = \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j p_{t+j}^t (\ell_{t+j}^t - \ell_{t+j}^0)$$

This version is more convenient for computation.

Using the equation

$$p_{t+j}^t = p_{t+1}^t p_{t+j}^{t+1}$$

it is possible to verify that (13) implies that

$$B_t = (\tau_t \ell_t - g_t) + E_t \sum_{j=1}^{\infty} p_{t+j}^t (\tau_{t+j} \ell_{t+j}^t - g_{t+j})$$

and

$$B_t = (\tau_t \ell_t - g_t) + \beta E_t p_{t+1}^t B_{t+1}$$

Define

$$R_t^{-1} := \mathbb{E}_t \beta^j p_{t+1}^t$$

$R_t$ is the gross 1-period risk-free rate for loans between $t$ and $t + 1$.

### 61.3.12 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 $\pi_t$

$$\Pi_t := \sum_{s=0}^{t} \pi_s$$

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 (14) and (15) to obtain

\[
\pi_{t+1} = B_{t+1} - \frac{1}{\beta E_p t_{t+1}} [\beta E_p t_{t+1} B_{t+1}]
\]

or

\[
\pi_{t+1} = B_{t+1} - \tilde{E}_t B_{t+1} \tag{16}
\]

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

\[
m_{t+1} = \frac{p_{t+1}}{E_{t} p_{t+1}}
\]

It follows from equation (16) 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 [8], 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.

### 61.4 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:

In [3]: using QuantEcon, Plots, LinearAlgebra, Parameters
    gr(fmt = :png);

    abstract type AbstractStochProcess end

    struct ContStochProcess{T<: AbstractFloat} <: AbstractStochProcess
        A::Matrix{T}
        C::Matrix{T}
    end
struct DiscreteStochProcess{TF <: AbstractFloat} <: AbstractStochProcess
  P::Matrix{TF}
  x_vals::Matrix{TF}
end

struct Economy{TF <: AbstractFloat, SP <: AbstractStochProcess}
  β::TF
  Sg::Matrix{TF}
  Sd::Matrix{TF}
  Sb::Matrix{TF}
  Ss::Matrix{TF}
  proc::SP
end

function compute_exog_sequences(econ, x)
  # compute exogenous variable sequences
  Sg, Sd, Sb, Ss = econ.Sg, econ.Sd, econ.Sb, econ.Ss
  g, d, b, s = [dropdims(S * x, dims = 1) for S in (Sg, Sd, Sb, Ss)]
  g, d, b, s, Sm
end

function compute_allocation(econ, Sm, ν, x, b)
  Sg, Sd, Sb, Ss = econ.Sg, econ.Sd, econ.Sb, econ.Ss
  Sc = 0.5 .* (Sb + Sd - Sg - ν .* Sm)
  Sl = 0.5 .* (Sb - Sd + Sg - ν .* Sm)
  c = dropdims(Sc * x, dims = 1)
  l = dropdims(Sl * x, dims = 1)
  p = dropdims((Sb - Sc) * x, dims = 1) # Price without normalization
  τ = 1 .- l ./ (b .- c)
  return Sc, Sl, c, l, p, τ, rvn
end

function compute_ν(a0, b0)
  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
61.4. IMPLEMENTATION

if ν * (0.5 - ν) < 0
    print("Negative multiplier on the government budget constraint.")
    error("Government spending (economy.g) too low")
end

return ν
end

function compute_Π(B, R, rvn, g, ξ)
    Π = cumsum(π .* ξ)
    return π, Π
end

function compute_paths(econ::Economy{<:AbstractFloat, <:DiscreteStochProcess}, T)
    # simplify notation
    @unpack β, Sg, Sd, Sb, Ss = econ
    @unpack P, x_vals = econ.proc

    mc = MarkovChain(P)
    state = simulate(mc, T, init=1)
    x = x_vals[:, state]

    # Compute exogenous sequence
    g, d, b, s, Sm = compute_exog_sequences(econ, x)

    # compute a0, b0
    ns = size(P, 1)
    F = I - 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
    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 = ((Sb - Sc) * x_vals) .* ((Sl - Sg) * x_vals) - (Sl * x_vals).^2
    temp = dropdims(F * H', dims = 2)
    B = temp[state] ./ p
    H = dropdims(P[state, :] * ((Sb - Sc) * x_vals)', dims = 2)
    R = p ./ (β .* H)
    temp = dropdims(P[state, :] *((Sb - Sc) * x_vals)', dims = 2)
    ξ = p[2:end] ./ temp[1:end-1]

    # compute π
    π, Π = compute_Π(B, R, rvn, g, ξ)

    return (g = g, d = d, b = b, s = s, c = c, l = l, p = p, τ = τ, rvn = rvn, B = B, R = R, π = π, Π = Π, ξ = ξ)
end
function compute_paths(econ::Economy{<:AbstractFloat, <:ContStochProcess}, T)
    # simplify notation
    @unpack β, Sg, Sd, Sb, Ss = econ
    @unpack A, C = econ.proc

    # generate an initial condition x₀ satisfying x₀ = A x₀
    nx, nx = size(A)
    x₀ = nullspace(I - A)
    x₀[end] < 0 ? -x₀ : x₀
    x₀ = x₀ ./ x₀[end]
    x₀ = dropdims(x₀, dims = 2)

    # generate a time series x of length T starting from x₀
    nw = size(C)
    w = randn(nw, T)
    x[:, 1] = x₀
    for t in 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)'*(Sl - Sg)
    L = zeros(T)
    for t in eachindex(L)
        L[t] = var_quadratic_sum(A, C, H, β, x[:, t])
    end
    B = L ./ p
    Rinv = dropdims(β .* (Sb - Sc)*A*x, dims = 1) ./ p
    R = 1 ./ Rinv
    AF1 = (Sb - Sc) * x[:, 2:end]
    AF2 = (Sb - Sc) * A * x[:, 1:end-1]
    ξ = AF1 ./ AF2
    ξ = dropdims(ξ, dims = 1)

    # compute π
    π, II = compute_II(B, R, rvn, g, ξ)

    return (g = g, d = d, b = b, s = s, c = c, l = l, p = p, τ = τ, rvn = rvn, B = B, R = R,
The function `var_quadratic_sum` from `QuantEcon.jl` is for computing the value of (11) 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.
61.5 Examples

Let’s look at two examples of usage.

61.5.1 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

```plaintext
In [4]: # for reproducible results
using Random
Random.seed!(42)

# parameters
\( \beta = 1 / 1.05 \)
\( \rho, mg = .7, .35 \)
\[ A = [\rho mg (1 - \rho); 0.0 1.0] \]
\[ C = [sqrt(1 - \rho^2) * mg / 10 0.0; 0 0] \]
\( Sg = [1.0 0.0] \)
\( Sd = [0.0 0.0] \)
\( Sb = [0 2.135] \)
\( Ss = [0.0 0.0] \)
proc = ContStochProcess(A, C)
econ = Economy(\( \beta \), Sg, Sd, Sb, Ss, proc)
T = 50
path = compute_paths(econ, T)
gen_fig_1(path)
```

Out[4]:
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.

In [5]: gen_fig_2(path)

Out[5]:
### 61.5.2 The Discrete Case

Our second example adopts a discrete Markov specification for the exogenous process

In [6]: # Parameters
β = 1 / 1.05
P = 
[0.8 0.2 0.0
 0.0 0.5 0.5
 0.0 0.0 1.0]

# Possible states of the world
# Each column is a state of the world. The rows are [g d b s 1]
x_vals = 
[0.5 0.5 0.25;
 0.0 0.0 0.0;
 2.2 2.2 2.2;
 0.0 0.0 0.0;
 1.0 1.0 1.0]

Sg = 
[1.0 0.0 0.0 0.0 0.0]
Sd = 
[0.0 1.0 0.0 0.0 0.0]
Sb = 
[0.0 0.0 1.0 0.0 0.0]
Ss = 
[0.0 0.0 0.0 1.0 0.0]
proc = DiscreteStochProcess(P, x_vals)

econ = Economy(β, Sg, Sd, Sb, Ss, proc)
T = 15
path = compute_paths(econ, Τ)

gen_fig_1(path)

Out[6]:

The call `gen_fig_2(path)` generates

In [7]: gen_fig_2(path)

Out[7]:

```python
path = compute_paths(econ, Τ)
gen_fig_1(path)
```
61.6 Exercises

61.6.1 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.

61.7 Solutions

In [8]: # parameters
\[
\beta = 1 / 1.05 \\
\rho, \text{ mg} = .95, .35 \\
A = [0. 0. 0. \rho \text{ mg}^*(1-\rho); \\
1. 0. 0. 0. 0.; \\
0. 1. 0. 0. 0.; \\
0. 0. 1. 0. 0.; \\
\]
\[
C = \text{zeros}(5, 5)
\]
\[
C[1, 1] = \text{sqrt}(1 - \rho^2) \times \frac{mg}{8}
\]
\[
S_g = [1. 0. 0. 0. 0.]
\]
\[
S_d = [0. 0. 0. 0. 0.]
\]
\[
S_b = [0. 0. 0. 0. 2.135]
\]
\[
S_s = [0. 0. 0. 0. 0.]
\]
\[
\text{proc} = \text{ContStochProcess}(A, C)
\]
\[
\text{econ} = \text{Economy}(\beta, S_g, S_d, S_b, S_s, \text{proc})
\]

\[
T = 50
\]
\[
\text{path} = \text{compute_paths}(\text{econ}, T)
\]

\text{In [9]:} \text{gen\_fig\_1(path)}

\text{Out[9]:}

\text{In [10]:} \text{gen\_fig\_2(path)}

\text{Out[10]:}
Chapter 62

Optimal Taxation with State-Contingent Debt

62.1 Contents

- Overview 62.2
- A Competitive Equilibrium with Distorting Taxes 62.3
- Recursive Formulation of the Ramsey problem 62.4
- Examples 62.5
- Further Comments 62.6

62.2 Overview

This lecture describes a celebrated model of optimal fiscal policy by Robert E. Lucas, Jr., and Nancy Stokey [72].

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.

62.2.1 Setup

```julia
In [1]: using InstantiateFromURL
"# optionally add arguments to force installation: instantiate = true,"
"precompile = true"
github_project("QuantEcon/quantecon-notesbooks-julia", version = "0.8.0")
```

```julia
In [2]: using LinearAlgebra, Statistics
using QuantEcon, NLsolve, NLopt, Interpolations
```

62.3 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 \( n_t \):

\[
\ell_t(s^t) + n_t(s^t) = 1 \tag{1}
\]

Output equals \( n_t(s^t) \) and can be divided between \( c_t(s^t) \) and \( g_t(s^t) \)

\[
c_t(s^t) + g_t(s^t) = n_t(s^t) \tag{2}
\]

A representative household’s preferences over \( \{c_t(s^t), \ell_t(s^t)\}_{t=0}^{\infty} \) are ordered by

\[
\sum_{t=0}^{\infty} \sum_{s^t} \beta^t \pi_t(s^t) u[c_t(s^t), \ell_t(s^t)] \tag{3}
\]

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
62.3. A COMPETITIVE EQUILIBRIUM WITH DISTORTING TAXES

\[ g_t(s^t) = \tau_t(s^t)n_t(s^t) + \sum_{s_{t+1}} p_{t+1}(s_{t+1}|s^t)b_{t+1}(s_{t+1}|s^t) - b_t(s_t|s^{t-1}) \]  \quad (4)

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

\[ c_t(s^t) + \sum_{s_{t+1}} p_t(s_{t+1}|s^t)b_{t+1}(s_{t+1}|s^t) = [1 - \tau_t(s^t)]n_t(s^t) + b_t(s_t|s^{t-1}) \quad \forall t \geq 0. \]  \quad (5)

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 (2) 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 (3) subject to (5) and (1) 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 household’s optimization problem.
- Given the allocation, government policy, and price system, the government’s budget constraint is satisfied for all \( t, s^t \).

Note: There 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 (3).

62.3.1 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 \( q_t^0(s^t) \) be the price at time 0, measured in time 0 consumption goods, of one unit of consumption at time \( t \), history \( s^t \).

The following recursion relates Arrow-Debreu prices \( \{q_t^0(s^t)\}_{t=0}^{\infty} \) to Arrow securities prices \( \{p_{t+1}(s_{t+1}|s^t)\}_{t=0}^{\infty} \)

\[ q_{t+1}^0(s^{t+1}) = p_{t+1}(s_{t+1}|s^t)q_t^0(s^t) \quad s.t. \quad q_0^0(s^0) = 1 \]  \quad (6)
CHAPTER 62. OPTIMAL TAXATION WITH STATE-CONTINGENT DEBT

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.

62.3.2 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 household’s problem and solve them for \( \{q_t^0(s^t), \tau_t(s^t)\}_{t=0}^{\infty} \) as functions of the allocation \( \{c_t(s^t), n_t(s^t)\}_{t=0}^{\infty} \).

2. Substitute these expressions for taxes and prices in terms of the allocation into the household’s present-value budget constraint.

   - This intertemporal constraint involves only the allocation and is regarded as an implementability constraint.

3. Find the allocation that maximizes the utility of the representative household (3) subject to the feasibility constraints (1) and (2) 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.

62.3.3 The Implementability Constraint

By sequential substitution of one one-period budget constraint (5) into another, we can obtain the household’s present-value budget constraint:

\[
\sum_{t=0}^{\infty} \sum_{s^t} q_t^0(s^t)c_t(s^t) = \sum_{t=0}^{\infty} \sum_{s^t} q_t^0(s^t)[1 - \tau_t(s^t)]n_t(s^t) + b_0
\]  \( (7) \)

\( \{q_t^0(s^t)\}_{t=1}^{\infty} \) can be interpreted as a time 0 Arrow-Debreu price system.

To approach the Ramsey problem, we study the household’s optimization problem. First-order conditions for the household’s problem for \( \ell_t(s^t) \) and \( b_t(s_{t+1} | s^t) \), respectively, imply

\[
(1 - \tau_t(s^t)) = \frac{u_t(s^t)}{u_c(s^t)}
\]  \( (8) \)
and

\[ p_{t+1}(s_{t+1}|s^t) = \beta \pi(s_{t+1}|s^t) \left( \frac{u_c(s^t)}{u_c(s^0)} \right) \]  

(9)

where \( \pi(s_{t+1}|s^t) \) is the probability distribution of \( s_{t+1} \) conditional on history \( s^t \).

Equation (9) implies that the Arrow-Debreu price system satisfies

\[ q_0^0(s^t) = \beta \pi_t(s^t) \frac{u_c(s^t)}{u_c(s^0)} \]  

(10)

Using the first-order conditions (8) and (9) to eliminate taxes and prices from (7), we derive the implementability condition

\[ \sum_{t=0}^{\infty} \sum_{s^t} \beta^t \pi_t(s^t)[u_c(s^t)c_t(s^t) - u_c(s^t)n_t(s^t)] - u_c(s^0)b_0 = 0. \]  

(11)

The Ramsey problem is to choose a feasible allocation that maximizes

\[ \sum_{t=0}^{\infty} \sum_{s^t} \beta^t \pi_t(s^t)u[c_t(s^t), 1 - n_t(s^t)] \]  

subject to (11).

62.3.4 Solution Details

First define a “pseudo utility function”

\[ V[c_t(s^t), n_t(s^t), \Phi] = u[c_t(s^t), 1 - n_t(s^t)] + \Phi [u_c(s^t)c_t(s^t) - u_c(s^t)n_t(s^t)] \]  

(13)

where \( \Phi \) is a Lagrange multiplier on the implementability condition (7).

Next form the Lagrangian

\[ J = \sum_{t=0}^{\infty} \sum_{s^t} \beta^t \pi_t(s^t) \left\{ V[c_t(s^t), n_t(s^t), \Phi] + \theta_t(s^t) \left[ n_t(s^t) - c_t(s^t) - g_t(s^t) \right] \right\} - \Phi u_c(0)b_0 \]  

(14)

where \( \{\theta_t(s^t); \forall s^t\}_{t \geq 0} \) is a sequence of Lagrange multipliers on the feasible conditions (2).

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

\[ c_t(s^t): (1 + \Phi)u_c(s^t) + \Phi [u_{cc}(s^t)c_t(s^t) - u_{cc}(s^t)n_t(s^t)] - \theta_t(s^t) = 0, \quad t \geq 1 \]

\[ n_t(s^t): - (1 + \Phi)u_{\ell}(s^t) - \Phi [u_{\ell\ell}(s^t)c_t(s^t) - u_{\ell\ell}(s^t)n_t(s^t)] + \theta_t(s^t) = 0, \quad t \geq 1 \]  

(15)

and
\[ c_0(s^0, b_0): (1 + \Phi)u_c(c, 1 - c - g) + \Phi [u_{cc}(c, 1 - c - g) - (c + g)u_{ct}(c, 1 - c - g)] = (1 + \Phi)u_{c\ell}(c, 1 - c - g) + \Phi [u_{cc}(c, 1 - c - g) - (c + g)u_{ct}(c, 1 - c - g)] + \Phi (u_{cc} - u_{c,\ell})b_0 \]

Please note how these first-order conditions differ between \( t = 0 \) and \( t \geq 1 \).

It is instructive to use first-order conditions (16) 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

\[ (1 + \Phi)u_c(c, 1 - c - g) + \Phi [u_{cc}(c, 1 - c - g) - (c + g)u_{ct}(c, 1 - c - g)] = (1 + \Phi)u_{\ell}(c, 1 - c - g) + \Phi [u_{cc}(c, 1 - c - g) - (c + g)u_{\ell}(c, 1 - c - g)] \]

where we have imposed conditions (1) and (2).

Equation (17) is one equation that can be solved to express the unknown \( c \) as a function of the exogenous variable \( g \).

We also know that time \( t = 0 \) quantities \( c_0 \) and \( n_0 \) satisfy

\[ (1 + \Phi)u_c(c, 1 - c - g) + \Phi [u_{cc}(c, 1 - c - g) - (c + g)u_{ct}(c, 1 - c - g)] = (1 + \Phi)u_{\ell}(c, 1 - c - g) + \Phi [u_{cc}(c, 1 - c - g) - (c + g)u_{\ell}(c, 1 - c - g)] + \Phi (u_{cc} - u_{c,\ell})b_0 \]

Notice that a counterpart to \( b_0 \) does not appear in (17), 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 (16) 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

\[ g_t(s^t) = g_t(\tilde{s}^\tau) = g \]

then it follows from (17) that the Ramsey choices of consumption and leisure, \((c_t(s^t), \ell_t(s^t))\) and \((c_{\tau}(\tilde{s}^\tau), \ell_{\tau}(\tilde{s}^\tau))\), 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 \( g \).

**62.3.5 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 (16).
Evidently, for \( t \geq 1 \), \( c \) and \( n \) depend on the time \( t \) realization of \( g \) only.

But for \( t = 0 \), \( c \) and \( n \) depend on both \( g_0 \) and the government’s initial debt \( b_0 \).

Thus, while \( b_0 \) influences \( c_0 \) and \( n_0 \), there appears no analogous variable \( b_t \) that influences \( c_t \) and \( n_t \) for \( t \geq 1 \).

The absence of \( b_t \) as a determinant of the Ramsey allocation for \( t \geq 1 \) and its presence for \( t = 0 \) is a symptom of the \emph{time-inconsistency} of a Ramsey plan.

\( \Phi \) has to take a value that assures that the household and the government’s budget constraints are both satisfied at a candidate Ramsey allocation and price system associated with that \( \Phi \).

**62.3.6 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.

**62.3.7 Determining \( \Phi \)**

We complete the Ramsey plan by computing the Lagrange multiplier \( \Phi \) on the implementability constraint (11).

Government budget balance restricts \( \Phi \) via the following line of reasoning.

The household’s first-order conditions imply

\[
(1 - \tau_t(s^t)) = \frac{u_t(s^t)}{u_c(s^t)} \quad (19)
\]

and the implied one-period Arrow securities prices

\[
p_{t+1}(s_{t+1}|s^t) = \beta \Pi(s_{t+1}|s_t) \frac{u_c(s^{t+1})}{u_c(s^t)} \quad (20)
\]

Substituting from (19), (20), and the feasibility condition (2) into the recursive version (5) of the household budget constraint gives

\[
u_c(s^t)[n_t(s^t) - g_t(s^t)] + \beta \sum_{s_{t+1}} \Pi(s_{t+1}|s_t) u_c(s^{t+1}) b_{t+1}(s_{t+1}|s^t) = u_t(s^t)n_t(s^t) + u_c(s^t)b_t(s_t|s^{t-1}) \quad (21)
\]

Define \( x_t(s^t) = u_c(s^t)b_t(s_t|s^{t-1}) \).
Notice that \(y_t(s_t)\) appears on the right side of (21) while \(\beta\) times the conditional expectation of \(y_{t+1}(s_{t+1})\) 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 (21) as

\[
u_c(s)[n(s) - g(s)] + \beta \sum_{s'} \Pi(s'|s)x'(s') = u_l(s)n(s) + x(s)
\]

where \(s'\) denotes a next period value of \(s\) and \(x'(s')\) denotes a next period value of \(x\).

Equation (22) is easy to solve for \(x(s)\) for \(s = 1, \ldots, S\).

If we let \(\tilde{n}, \tilde{g}, \tilde{x}\) denote \(S \times 1\) vectors whose \(i\)th elements are the respective \(n, g, x\) values when \(s = i\), and let \(\Pi\) be the transition matrix for the Markov state \(s\), then we can express (22) as the matrix equation

\[
\tilde{u}_c(\tilde{n} - \tilde{g}) + \beta \Pi \tilde{x} = \tilde{u}_l \tilde{n} + \tilde{x}
\]

This is a system of \(S\) linear equations in the \(S \times 1\) vector \(x\), whose solution is

\[
x = (I - \beta \Pi)^{-1}[\tilde{u}_c(\tilde{n} - \tilde{g}) - \tilde{u}_l \tilde{n}]
\]

In these equations, by \(\tilde{u}_c \tilde{n}\), for example, we mean element-by-element multiplication of the two vectors.

After solving for \(\tilde{x}\), we can find \(b_t(s_t|s_{t-1})\) in Markov state \(s_t = s\) from \(b(s) = \frac{x(s)}{u_c(s)}\) or the matrix equation

\[
\tilde{b} = \frac{\tilde{x}}{\tilde{u}_c}
\]

where division here means element-by-element division of the respective components of the \(S \times 1\) vectors \(\tilde{x}\) and \(\tilde{u}_c\).

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 (24) for the \(S\) elements of \(\tilde{x}\)
   - these depend on \(\Phi\)
3. Find a \(\Phi\) that satisfies
\[ u_{c,0}b_0 = u_{c,0}(n_0 - g_0) - u_{t,0}n_0 + \beta \sum_{s=1}^{S} \Pi(s|s_0)x(s) \] (26)

by gradually raising \( \Phi \) if the left side of (26) exceeds the right side and lowering \( \Phi \) if the left side is less than the right side.

1. After computing a Ramsey allocation, recover the flat tax rate on labor from (8) and the implied one-period Arrow securities prices from (9).

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 \( \Phi \).

62.3.8 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 \( \{\tau_j(s^t)\}_{j=t}^{\infty}, \{b_{j+1}(s_{j+1}|s^t)\}_{j=t}^{\infty} \) is a time \( t \), history \( s^t \) continuation of a time 0, state \( s_0 \) Ramsey plan.

A time \( t \), history \( s^t \) Ramsey plan is a Ramsey plan that starts from initial conditions \( s^t, b_t(s_t|s^{t-1}) \).

A time \( t \), history \( s^t \) continuation of a time 0, state \( s^0 \) Ramsey plan is not a time \( t \), history \( s^t \) Ramsey plan.

The means that a Ramsey plan is not time consistent.

Another way to say the same thing is that a Ramsey plan is time inconsistent.

The reason is that a continuation Ramsey plan takes \( u_{c,t}b_t(s_t|s^{t-1}) \) as given, not \( b_t(s_t|s^{t-1}) \).

We shall discuss this more below.

62.3.9 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) \[2\], 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 \[72\] 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

\[
\begin{align*}
  u_\ell(c, \ell) &\sim -u_n(c, n) \\
  u_c(c, \ell) &\sim u_n(c, n) \\
  u_{\ell,\ell}(c, \ell) &\sim u_{nn}(c, n) \\
  u_{c,c}(c, \ell) &\sim u_{c,\ell}(c, n) \\
  u_{c,\ell}(c, \ell) &\sim 0
\end{align*}
\]

With these understandings, equations (17) and (18) simplify in the case of the CRRA utility function.
They become

\[
(1 + \Phi)[u_c(c) + u_n(c + g)] + \Phi[u_{ee}(c) + (c + g)u_{nn}(c + g)] = 0 \tag{27}
\]

and

\[
(1 + \Phi)[u_c(c_0) + u_n(c_0 + g_0)] + \Phi[c_0u_{ee}(c_0) + (c_0 + g_0)u_{nn}(c_0 + g_0)] - \Phi u_{cc}(c_0)b_0 = 0 \tag{28}
\]

In equation (27), 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 \( b_0 \):

\[
b_0 + g_0 = \tau_0(c_0 + g_0) + \frac{\bar{b}}{R_0} \tag{29}
\]

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 (29), it is understood that

\[
\tau_0 = 1 - \frac{u_{\ell,0}}{u_{c,0}}
\]

\[
R_0 = \beta \sum_{s=1}^{S} \Pi(s | s_0) \frac{u_c(s)}{u_{c,0}}
\]

62.3.10 Sequence Implementation

The above steps are implemented in a type called SequentialAllocation

```python
In [3]: using QuantEcon, NLsolve, NLopt, LinearAlgebra, Interpolations
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
    Un::Function
    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)
    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, S, version)
    if version !*= 1 && version !*= 2
        throw(ArgumentError("version must be 1 or 2"))
    end
    β, Ω, 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] = Ω .* Uc(c, n) + Un(c, n)
        out[S+1:end] = Ω .* 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
\[
\begin{align*}
\text{cFB} &= \text{res.zero}[1:S] \\
nFB &= \text{res.zero}[S+1:end] \\
Ξ &= \text{Uc}(\text{cFB}, \text{nFB}) \\
zFB &= \text{vcat}(\text{cFB}, \text{nFB}, Ξ) \\
\text{return} \ (\text{cFB}, \text{nFB}, Ξ, zFB)
\end{align*}
\]

\text{elseif} \ \text{version} == 2
\begin{align*}
\text{cFB} &= \text{res.zero}[1:S] \\
nFB &= \text{res.zero}[S+1:end] \\
Ξ &= \text{Uc}(\text{cFB}, \text{nFB}).\text{cFB} + \text{Un}(\text{cFB}, \text{nFB}).\text{nFB} \\
xFB &= (\text{I} - β * \text{II}, \text{IFB}) \\
zFB &= \text{vcat}(\text{cFB}[s], \text{xFB}[s], \text{xFB}) \text{ for } s \text{ in } 1:S \\
\text{return} \ (\text{cFB}, \text{nFB}, Ξ, xFB, zFB)
\end{align*}

\text{end}

\text{end}

\text{function} \ \text{time1\_allocation}(p: SequentialAllocation, μ)
\begin{align*}
\text{model, S} &= p.\text{model, p.S} \\
Θ, β, II, G &= \text{model.Θ, model.β, model.II, model.G} \\
\text{model.Uc, model.Ucc, model.Un, model.Unn} &= \text{model.Uc, model.Ucc, model.Un, model.Unn}
\end{align*}

\text{function} \ \text{FOC!}(\text{out, z})
\begin{align*}
c &= z[1:S] \\
n &= z[S+1:2S] \\
Ξ &= z[2S+1:end] \\
\text{out}[1:S] &= \text{Uc}(c, n) - μ * (\text{Ucc}(c, n) .* c + \text{Uc}(c, n)) - Ξ \text{ # FOC c} \\
\text{out}[S+1:2S] &= \text{Un}(c, n) - μ * (\text{Unn}(c, n) .* n + \text{Un}(c, n)) + Θ .* Ξ \text{ # FOC n}
\end{align*}

\text{return} \ \text{out}
\]

\text{end}

# Find the root of the FOC
\text{res} = \text{nlsolve(FOC!, p.zFB)}
\text{if} \ \text{res.f\_converged} == \text{false}
\text{error("Could not find LS allocation.")}
\text{end}

z = \text{res.zero}
\begin{align*}
c, n, Ξ &= z[1:S], z[S+1:2S], z[2S+1:end] \\
\text{Inv} &= \text{Uc}(c, n) .* c + \text{Un}(c, n) .* n \\
x &= (\text{I} - β * \text{model.II, Inv}) \\
\text{return} \ c, n, x, Ξ
\end{align*}

\text{end}

\text{function} \ \text{time0\_allocation}(p: SequentialAllocation, B_, s_0)
\begin{align*}
\text{model} &= p.\text{model} \\
\text{II, Θ, G, β} &= \text{model.II, model.Θ, model.G, model.β} \\
\text{Uc, Ucc, Un, Unn} &= \text{model.Uc, model.Ucc, model.Un, model.Unn}
\end{align*}

# First order conditions of planner's problem
\text{function} \ \text{FOC!}(\text{out, z})
\begin{align*}
\mu, c, n, Ξ &= z[1], z[2], z[3], z[4] \\
xprime &= \text{time1\_allocation}(\text{pas, μ})[3] \\
\text{out} &= \text{vcat}(\text{Uc}(c, n) .* (c - B_) + \text{Un}(c, n) .* n + β * \text{dot(II[s_0, :], xprime),} \\
\text{Uc}(c, n) - μ * (\text{Ucc}(c, n) .* (c - B_) + \text{Uc}(c, n)) - Ξ, \\
\text{Un}(c, n) - μ * (\text{Unn}(c, n) .* n + \text{Un}(c, n)) + Θ[s_0] .* Ξ,
\end{align*}

\text{return} \ \text{out}
(Θ .^ n .- c - G)[s_0]
)

# 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, μ)
    model = pas.model
    c, n, x, Ξ = time1_allocation(pas, μ)
    U_val = model.U.(c, n)
    V = \( I - \text{model.β} \ast \text{model.II, U_val} \)
    return c, n, x, V
end

function T(model, c, n)
    Uc, Un = model.Uc.(c, n), model.Un.(c, n)
    return 1 .+ Un ./ (model.Θ .^ Uc)
end

function simulate(pas::SequentialAllocation, B_, s_0, T, sHist = nothing)
    model = pas.model
    II, β, Uc = model.II, model.β, model.Uc
    if isnothing(sHist)
        sHist = QuantEcon.simulate(pas.mc, T, init=s_0)
    end
    cHist = zeros(T)
    nHist = similar(cHist)
    Bhist = similar(cHist)
    THist = similar(cHist)
    μHist = similar(cHist)
    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(II[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 cHist, nHist, Bhist, THist, sHist, μHist, RHist
end
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

function BellmanEquation(model, xgrid, policies0)
    S = size(model.Π, 1) # Number of states
    xbar = collect(extrema(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, i_x, x, s, Vf)
    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[s]
        Vprime = [Vf[sprime](xprime[sprime]) for sprime in 1:S]
        return -(U(c, n) + β * dot(Π[s, :], Vprime))
    end

    function cons(z, 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]
        end
    end

end
62.4 Recursive Formulation of the Ramsey problem

\[ x_t(s^t) = u_t(s^t) b_t(s_t|s^{t-1}) \] in equation (21) 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.
CHAPTER 62. OPTIMAL TAXATION WITH STATE-CONTINGENT DEBT

62.4.1 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.

62.4.2 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 \( W(b, s) \).

62.4.3 The Continuation Ramsey Problem

The Bellman equation for a time \( t \geq 1 \) continuation Ramsey planner is

\[
V(x, s) = \max_{n, \{x'(s')\}} u(n - g(s), 1 - n) + \beta \sum_{s' \in S} \Pi(s'|s) V(x', s')
\]  

(30)

where maximization over \( n \) and the \( S \) elements of \( x'(s') \) is subject to the single implementability constraint for \( t \geq 1 \)

\[
x = u_c(n - g(s)) - u_l n + \beta \sum_{s' \in S} \Pi(s'|s) x'(s')
\]  

(31)
62.4. RECURSIVE FORMULATION OF THE RAMSEY PROBLEM

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 (30) are \( S + 1 \) time-invariant policy functions

\[
n_t = f(x_t, s_t), \quad t \geq 1
\]

\[
x_{t+1}(s_{t+1}) = h(s_{t+1}; x_t, s_t), \quad s_{t+1} \in S, \quad t \geq 1
\]  

(32)

62.4.4 The Ramsey Problem

The Bellman equation for the time 0 Ramsey planner is

\[
W(b_0, s_0) = \max_{n_0, \{x(s_1)\}} u(n_0 - g_0, 1 - n_0) + \beta \sum_{s_1 \in S} \Pi(s_1|s_0)V(x'(s_1), s_1)
\]  

(33)

where maximization over \( n_0 \) and the \( S \) elements of \( x'(s_1) \) is subject to the time 0 implementability constraint

\[
u_{c,0}b_0 = u_{c,0}(n_0 - g_0) - u_{l,0}n_0 + \beta \sum_{s_1 \in S} \Pi(s_1|s_0)x'(s_1)
\]  

(34)

coming from restriction (26).

Associated with a value function \( W(b_0, n_0) \) that solves Bellman equation (33) are \( S + 1 \) time 0 policy functions

\[
n_0 = f_0(b_0, s_0)
\]

\[
x_1(s_1) = h_0(s_1; b_0, s_0)
\]  

(35)

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 (32) for the time \( t \geq 1 \) continuation Ramsey planners.

The value function \( V(x_t, s_t) \) of the time \( t \) continuation Ramsey planner equals

\[
E_t \sum_{\tau=t}^{\infty} \beta^{\tau-t}u(c_\tau, l_\tau), \quad \text{where the consumption and leisure processes are evaluated along the original time 0 Ramsey plan.}
\]

62.4.5 First-Order Conditions

Attach a Lagrange multiplier \( \Phi_1(x, s) \) to constraint (31) and a Lagrange multiplier \( \Phi_0 \) to constraint (26).

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 planner’s Bellman equation (30) are

\[
\beta \Pi(s'|s)V_x(x', s') - \beta \Pi(s'|s)\Phi_1 = 0
\]  

(36)

for \( x'(s') \) and
\[(1 + \Phi_1)(u_c - u_t) + \Phi_1 \left[ n(u_L - u_{lc}) + (n - g(s))(u_{cc} - u_{lc}) \right] = 0 \] 

(37)

for \( n \).

Given \( \Phi_1 \), equation (37) is one equation to be solved for \( n \) as a function of \( s \) (or of \( g(s) \)).

Equation (36) implies \( V_x'(x', s') = \Phi_1 \), while an envelope condition is \( V_x'(x, s) = \Phi_1 \), so it follows that

\[ V_x'(x', s') = V_x'(x, s) = \Phi_1(x, s) \] 

(38)

Time \( t = 0 \): For the time 0 problem on the right side of the Ramsey planner’s Bellman equation (33), first-order conditions are

\[ V_x(x(s_1), s_1) = \Phi_0 \] 

(39)

for \( x(s_1), s_1 \in S \), and

\[ (1 + \Phi_0)(u_{c,0} - u_{n,0}) + \Phi_0 \left[ n_0(u_{ll,0} - u_{lc,0}) + (n_0 - g(s_0))(u_{cc,0} - u_{cl,0}) \right] - \Phi_0(u_{cc,0} - u_{cl,0})b_0 = 0 \] 

(40)

Notice similarities and differences between the first-order conditions for \( t \geq 1 \) and for \( t = 0 \).

An additional term is present in (40) 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.

### 62.4.6 State Variable Degeneracy

Equations (39) and (40) imply that \( \Phi_0 = \Phi_1 \) and that

\[ V_x(x_t, s_t) = \Phi_0 \] 

(41)

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 (34) for \( n \) and \( c \) as functions of \( g \) that are associated with \( \Phi = \Phi_0 \).
62.4.7 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 planner’s incentive to manipulate Arrow security prices and, through them, the value of initial government debt $b_0$.

62.4.8 Recursive Implementation

The above steps are implemented in a type called RecursiveAllocation

In [4]:
```plaintext
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
```

```plaintext
function RecursiveAllocation(model, μgrid)
    mc = MarkovChain(model.Π)
    G = model.G
    S = size(model.Π, 1) # Number of states
    return RecursiveAllocation{model, TI}(mc, S)
end
```
# 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)

function solve_time1_bellman(model, μgrid)
    μgrid0 = μgrid
    S = size(model.Π, 1)
    # First get initial fit
    PP = SequentialAllocation(model)
c = zeros(length(μgrid), 2)
n = similar(c)
x = similar(c)
V = similar(c)
for (i, μ) in enumerate(μgrid0)
c[i, :], n[i, :], x[i, :], V[i, :] = time1_value(PP, μ)
end
Vf = Vector{AbstractInterpolation}(undef, 2)
cf = similar(Vf)
nf = similar(Vf)
xprimef = similar(Vf, 2, S)
for s in 1:2
    cf[s] = LinearInterpolation(x[:, s][end:-1:1], c[:, s][end:-1:1])
nf[s] = LinearInterpolation(x[:, s][end:-1:1], n[:, s][end:-1:1])
Vf[s] = LinearInterpolation(x[:, s][end:-1:1], V[:, s][end:-1:1])
    for spроме in 1:S
        xprimef[s, spроме] = LinearInterpolation(x[:, s][end:-1:1], x[:, s][end:-1:1])
    end
end
policies = [cf, nf, xprimef]
# Create xgrid
xbar = [maximum(minimum(x, dims = 1)), minimum(maximum(x, dims = 1))]
xgrid = range(xbar[1], xbar[2], length = length(μgrid0))
# 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))/Vf[s](xgrid)))
    end
    print("diff = $diff \n")
    Vf = Vfnew
end
# Store value function policies and Bellman Equations
return Vf, policies, T, xgrid
end

function fit_policy_function(PP, PF, xgrid)
S = PP.S
Vf = Vector_AbstractInterpolation(undef, S)
cf = similar(Vf)
f = similar(Vf)
xprimef = similar(Vf, S, S)
for s in 1:S
PFvec = zeros(length(xgrid), 3+S)
for (i_x, x) in enumerate(xgrid)
    PFvec[i_x, :] = PF(i_x, x, s)
end
Vf[s] = LinearInterpolation(xgrid, PFvec[:, 1])
cf[s] = LinearInterpolation(xgrid, PFvec[:, 2])
f[s] = LinearInterpolation(xgrid, PFvec[:, 3])
for sprime in 1:S
    xprimef[s, sprime] = LinearInterpolation(xgrid, PFvec[:, 0]
        -3+sprime})
end
end
return Vf, [cf, f, xprimef]
end

function time0_allocation(pab::RecursiveAllocation, B_, s0)
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
return c0, n0, xprime0
end

function simulate(pab::RecursiveAllocation, B_, s_0, T,
    model, S, policies = pab.model, pab.S, pab.policies
    β, II, Uc = model.β, model.II, model.Uc
    cf, f, xprimef = policies[1], policies[2], policies[3]
cHist = zeros(T)
    nHist = similar(cHist)
    Bhist = similar(cHist)
    THist = similar(cHist)
    μHist = similar(cHist)
    RHist = zeros(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]]
n = nf[s](x)
c = [cf[shat](x) for shat in 1:S]

xprime = [xprimef[s, sprime](x) for sprime in 1:S]

T[shist[t]] = T(pab.model, c, n)[s]
ung[c, n] = Uc(c, n)

\[ \Pi_{Hist[t]} = \mathcal{P}(\text{pab.model, c, n})[s] \]

\[ \mu_{Hist[t]} = \text{pab.Vf}[s](x) \]

\[ R_{Hist[t-1]} = \text{Uc}(\text{cHist[t-1], nHist[t-1]}) / (\beta \ast \text{Eu}_{c[s]} \]

\[ \text{chist[t], nHist[t], Bhist[t]} = c[s], n, x / \text{u}_{c[s]} \]

\[ \text{return chist, nHist, Bhist, THist, shist, mHist, RHist} \]

\textbf{Out}[4]: simulate (generic function with 7 methods)

\section{62.5 Examples}

\subsection{62.5.1 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

\[ g = \begin{pmatrix}
0.1 \\
0.1 \\
0.2 \\
0.1
\end{pmatrix} \]

We assume that the representative agent has utility function
62.5. EXAMPLES

\[ u(c, n) = \frac{c^{1-\sigma}}{1-\sigma} - \frac{n^{1+\gamma}}{1+\gamma} \]

and set \( \sigma = 2, \gamma = 2 \), and the discount factor \( \beta = 0.9 \).

Note: For convenience in terms of matching our code, we have expressed utility as a function of \( n \) rather than leisure \( l \).

This utility function is implemented in the type CRRAutility

In [5]: function crra_utility();
   \( \beta = 0.9 \),
   \( \sigma = 2.0 \),
   \( \gamma = 2.0 \),
   \( \Pi = 0.5 \times \text{ones}(2, \ 2) \),
   \( G = [0.1, \ 0.2] \),
   \( \Theta = \text{ones}(2) \),
   \( \text{transfers} = \text{false} \)
      )
   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.\(^{-\sigma - 1.0} \)
   Un(c,n) = -n.\(^{-\gamma} \)
   Unn(c,n) = -\gamma * n.\(^{\gamma - 1.0} \)
   n_less_than_one = false
   return Model(\( \beta \), \( \Pi \), \( G \), \( \Theta \), \( \text{transfers} \),
              U, Uc, Ucc, Un, Unn, n_less_than_one)
end

Out[5]: crra_utility (generic function with 1 method)

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 \( g_3 = .2 \)

In [6]: using Random
   Random.seed!(42) # For reproducible results.

   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
   \( \phi \)-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 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 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 Plots
gr(fmt=:png);
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]]

plots = plot(layout=(3, 2), size=(800, 600))
for i = 1:6
    plot!(plots[i], sim_seq_l_plot[i], color=:black, lw=2,
          marker=:circle, markersize=2, label="")
    plot!(plots[i], sim_seq_h_plot[i], color=:red, lw=2,
          marker=:circle, markersize=2, label="")
    plot!(plots[i], title=titles[i], grid=true)
end
plot(plots)
62.5. EXAMPLES

**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

\[ \text{In [7]: } \text{plot(sim_seq_l[end], color=black, lw=2, marker=circle, markersize=2, label="")} \]
\[ \text{plot!(sim_seq_h[end], color=red, lw=2, marker=circle, markersize=2, label="")} \]
\[ \text{plot!(title="Gross Interest Rate", grid=true)} \]

\[ \text{Out[7]: } \]

62.5.2 Government Saving

At time \( t = 0 \) the government evidently dissaves since \( b_1 > b_0 \).

- This is a consequence of it setting a lower tax rate at \( t = 0 \), implying more consumption at \( t = 0 \).

At time \( t = 1 \), the government evidently saves since it has set the tax rate sufficiently high to allow it to set \( b_2 < b_1 \).

- Its motive for doing this is that it anticipates a likely war at \( t = 3 \).

At time \( t = 2 \) the government trades state-contingent Arrow securities to hedge against war at \( t = 3 \).

- It purchases a security that pays off when \( g_3 = g_h \).
• It sells a security that pays off when \( g_3 = g_l \).
• 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.

### 62.5.3 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.

### 62.5.4 Time 0 and Time-Inconsistency

In the preceding example, the Ramsey tax rate at time 0 differs from its value at time 1.

To explore what is going on here, let’s simplify things by removing the possibility of war at time \( t = 3 \).

The Ramsey problem then includes no randomness because \( g_t = g_l \) for all \( t \).

The figure below plots the Ramsey tax rates and gross interest rates at time \( t = 0 \) and time \( t \geq 1 \) as functions of the initial government debt (using the sequential allocation solution and a CRRA utility function defined above)

```
In [8]: M2 = crra_utility(G=[0.15], Π=ones(1, 1), Θ=[1.0])
PP_seq_time0 = SequentialAllocation(M2) # solve sequential problem
B_vec = range(-1.5, 1.0, length=100)
taxpolicy = Matrix(hcat([simulate(PP_seq_time0, B_, 1, 2)[4] for B_ in B_vec]...))
interest_rate = Matrix(hcat([simulate(PP_seq_time0, B_, 1, 3)[end] for B_ in B_vec]...))
titles = ["Tax Rate" "Gross Interest Rate"]
labels = [["Time , t = 0", "Time , t >= 0"], ""]
plots = plot(layout=(2,1), size=(700,600))
for (i, series) in enumerate((taxpolicy, interest_rate))
    plot!(plots[i], B_vec, series, linewidth=2, label=labels[i])
    plot!(plots[i], title=titles[i], grid=true, legend=:topleft)
end
plot(plots)
```

Out[8]:

...
The figure indicates that if the government enters with positive debt, it sets a tax rate at \( t = 0 \) that is less than all later tax rates.

By setting a lower tax rate at \( t = 0 \), the government raises consumption, which reduces the value \( u_{c,0}b_0 \) of its initial debt.

It does this by increasing \( c_0 \) and thereby lowering \( u_{c,0} \).

Conversely, if \( b_0 < 0 \), the Ramsey planner sets the tax rate at \( t = 0 \) higher than in subsequent periods.

A side effect of lowering time \( t = 0 \) consumption is that it raises the one-period interest rate at time 0 above that of subsequent periods.

There are only two values of initial government debt at which the tax rate is constant for all \( t \geq 0 \).

The first is \( b_0 = 0 \)

- Here the government can’t use the \( t = 0 \) tax rate to alter the value of the initial debt.

The second occurs when the government enters with sufficiently large assets that the Ramsey planner can achieve first best and sets \( \tau_t = 0 \) for all \( t \).

It is only for these two values of initial government debt that the Ramsey plan is time-consistent.
Another way of saying this is that, except for these two values of initial government debt, a continuation of a Ramsey plan is not a Ramsey plan.

To illustrate this, consider a Ramsey planner who starts with an initial government debt $b_1$ associated with one of the Ramsey plans computed above.

Call $\tau_1^R$ the time $t = 0$ tax rate chosen by the Ramsey planner confronting this value for initial government debt government.

The figure below shows both the tax rate at time 1 chosen by our original Ramsey planner and what a new Ramsey planner would choose for its time $t = 0$ tax rate.

```plaintext
In [9]: # 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 = Matrix(hcat([simulate(PP_seq_time0, B1, 1, 1)[4] for B1 in B1_vec]...))

   plot(B_vec, taxpolicy[:, 2], linewidth=2, label="tau_1")
   plot!(B_vec, tau1_reset, linewidth=2, label="tau_1^R")
   plot!(title="Tax Rate", xlabel="Initial Government Debt", legend=:topleft, grid=true)
```

Out[9]:

The tax rates in the figure are equal for only two values of initial government debt.

### 62.5.5 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 \( U_{cc}c = -\sigma U_c \) and \( U_{nn}n = \gamma U_n \) to derive

\[
\frac{(1 + (1 - \sigma)\Phi)U_c}{(1 + (1 - \gamma)\Phi)U_n} = 1
\]

from the first-order conditions.

This equation immediately implies that the tax rate is constant.

For other preferences, the tax rate may not be constant.

For example, let the period utility function be

\[ u(c, n) = \log(c) + 0.69 \log(1 - n) \]

We will write a new constructor LogUtility to represent this utility function

```
In [10]: 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)
    Un(c,n) = -ψ ./ (1.0 .- n)
    Unn(c,n) = -ψ ./ (1.0 .- n).^2.0
    n_less_than_one = true
    return Model(β, Π, G, Θ, transfers,
                 U, Uc, Ucc, Un, Unn, n_less_than_one)
end
```

```
Out[10]: log_utility (generic function with 1 method)
```

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

```
In [11]: M1 = log_utility()
    μ_grid = range(-0.6, 0.0, length = 200)
    PP_seq = SequentialAllocation(M1)  # Solve sequential problem
    PP_bel = RecursiveAllocation(M1, μ_grid)  # Solve recursive problem
    T = 20
    sHist = [1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 1, 1, 1, 2, 2, 2, 2, 2, 2, 1]```
# Simulate
sim_seq = simulate(PP_seq, 0.5, 1, T, sHist)
sim_bel = simulate(PP_bel, 0.5, 1, T, sHist)

# Plot policies
sim_seq_plot = [sim_seq[1:4], M1.G[sHist], M1.Θ[sHist].*sim_seq[2]]
sim_bel_plot = [sim_bel[1:4], M1.G[sHist], M1.Θ[sHist].*sim_bel[2]]
titles = hcat("Consumption",
    "Labor Supply",
    "Government Debt",
    "Tax Rate",
    "Government Spending",
    "Output")
labels = [["Sequential", "Recursive"], ["", ""], ["", ""], ["", ""], ["", ""]
    ["", ""], ["", ""]]
plots = plot(layout=(3,2), size=(850,780))
for i = 1:6
    plot!(plots[i], sim_seq_plot[i], color=:black, lw=2, marker=:circle,
        markersize=2, label=labels[i][1])
    plot!(plots[i], sim_bel_plot[i], color=:blue, lw=2, marker=:xcross,
        markersize=2, label=labels[i][2])
    plot!(plots[i], title=titles[i], grid=true, legend=:topright)
end
plot(plots)

diff = 0.0003504611535379196
diff = 0.00015763906388123851
diff = 7.124337606645018e-5
diff = 3.2356917242389125e-5
diff = 1.4829540976261937e-5
diff = 6.9104341194283816e-6
diff = 3.323222470321399e-6
diff = 1.6879569419905608e-6
diff = 9.29342141847281e-7
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.

### 62.6 Further Comments


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.
Chapter 63

Optimal Taxation without State-Contingent Debt

63.1 Contents

- Overview 63.2
- Competitive Equilibrium with Distorting Taxes 63.3
- Recursive Version of AMSS Model 63.4
- Examples 63.5

63.2 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 [72].


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.

63.2.1 Setup

In [1]: using InstantiateFromURL
   # optionally add arguments to force installation: instantiate = true,
   ⊲precompile = true
   github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")

In [2]: using LinearAlgebra, Statistics
### 63.3 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 supply \( n_t \):

\[
\ell_t(s^t) + n_t(s^t) = 1 \tag{1}
\]

Output equals \( n_t(s^t) \) and can be divided between consumption \( c_t(s^t) \) and \( g(s_t) \)

\[
c_t(s^t) + g(s_t) = n_t(s^t) \tag{2}
\]

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

\[
\sum_{t=0}^\infty \sum_{s^t} \beta^t \pi_t(s^t)u[c_t(s^t), \ell_t(s^t)] \tag{3}
\]

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 [2] 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 [8] than they do in [72].

#### 63.3.1 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 Section ??.
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

$$b_t(s^{t-1}) = \tau_t^n(s^t)n_t(s^t) - g_t(s_t) - T_t(s^t) + \frac{b_{t+1}(s^t)}{R_t(s^t)}$$

$$= z(s^t) + \frac{b_{t+1}(s^t)}{R_t(s^t)}, \tag{4}$$

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 (4) yields:

$$b_t(s^{t-1}) = z(s^t) + \beta \sum_{s^{t+1}|s^t} \pi_{t+1}(s^{t+1}|s^t) \frac{u_c(s^{t+1})}{u_c(s^t)} b_{t+1}(s^t) \tag{5}$$

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 (5) 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 (5) by the right side of next period’s budget constraint (associated with a particular realization $s_{t+1}$) we get

$$b_t(s^{t-1}) = z(s^t) + \sum_{s^{t+1}|s^t} \beta \pi_{t+1}(s^{t+1}|s^t) \frac{u_c(s^{t+1})}{u_c(s^t)} \left[ z(s^{t+1}) + \frac{b_{t+2}(s^{t+1})}{R_{t+1}(s^{t+1})} \right]$$

After making similar repeated substitutions for all future occurrences of government indebtedness, and by invoking the natural debt limit, we arrive at:

$$b_t(s^{t-1}) = \sum_{j=0}^{\infty} \sum_{s^{t+j}|s^t} \beta^j \pi_{t+j}(s^{t+j}|s^t) \frac{u_c(s^{t+j})}{u_c(s^t)} z(s^{t+j}) \tag{6}$$

Now let’s

- substitute the resource constraint into the net-of-interest government surplus, and
• use the household’s first-order condition \(1 - \tau^t_i(s^t) = u_e(s^t) / u_e(s^t)\) to eliminate the labor tax rate

so that we can express the net-of-interest government surplus \(z(s^t)\) as

\[
z(s^t) = \left[1 - \frac{u_e(s^t)}{u_e(s^t)}\right] [c_t(s^t) + g_t(s_t)] - g_t(s_t) - T_t(s^t).
\] (7)

If we substitute the appropriate versions of right side of (7) for \(z(s^{t+j})\) into equation (6), we obtain a sequence of implementability constraints on a Ramsey allocation in an AMSS economy.

Expression (6) at time \(t = 0\) and initial state \(s^0\) was also an implementability constraint on a Ramsey allocation in a Lucas-Stokey economy:

\[
b_0(s^{-1}) = E_0 \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^j)}{u_c(s^0)} z(s^j)
\] (8)

Indeed, it was the only implementability constraint there.

But now we also have a large number of additional implementability constraints

\[
b_t(s^{t-1}) = E_t \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^{t+j})}{u_c(s^t)} z(s^{t+j})
\] (9)

Equation (9) must hold for each \(s^t\) for each \(t \geq 1\).

63.3.2 Comparison with Lucas-Stokey Economy

The expression on the right side of (9) 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.

63.3.3 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

\[
\max \left\{ c_t(s^t), b_{t+1}(s^t) \right\} E_0 \sum_{t=0}^{\infty} \beta^t u \left( c_t(s^t), 1 - c_t(s^t) - g_t(s_t) \right)
\]

where the maximization is subject to
63.3. **COMPETITIVE EQUILIBRIUM WITH DISTORTING TAXES**

\[
\mathbb{E}_0 \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^j)}{u_c(s^0)} z(s^j) \geq b_0(s^{-1}) \tag{10}
\]

and

\[
\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^{t+j})}{u_c(s^t)} z(s^{t+j}) = b_t(s^{t-1}) \quad \forall s^t \tag{11}
\]

given \(b_0(s^{-1})\).

**Lagrangian Formulation**

Let \(\gamma_0(s^0)\) be a nonnegative Lagrange multiplier on constraint (10).

As in the Lucas-Stokey economy, this multiplier is strictly positive when the government must resort to distortionary taxation; otherwise it equals zero.

A consequence of the assumption that there are no markets in state-contingent securities and that a market exists only in a risk-free security is that we have to attach stochastic processes \(\{\gamma_t(s^t)\}_{t=1}^{\infty}\) of Lagrange multipliers to the implementability constraints (11).

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 (11), 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 Section ??.

These features flow from the fact that the government cannot use state-contingent debt and therefore cannot allocate its indebtedness efficiently across future states.

**63.3.4 Some Calculations**

It is helpful to apply two transformations to the Lagrangian.

Multiply constraint (10) by \(u_c(s^0)\) and the constraints (11) by \(\beta^t u_c(s^t)\).

Then a Lagrangian for the Ramsey problem can be represented as
\[ J = \mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t \left\{ u(c_t(s^t), 1 - c_t(s^t) - g_t(s_t)) \right. \]
\[ + \gamma_t(s^t) \left[ \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j u_c(s^{t+j}) z(s^{t+j}) - u_c(s^t) b_t(s^{t-1}) \right] \]
\[ = \mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t \left\{ u(c_t(s^t), 1 - c_t(s^t) - g_t(s_t)) \right. \]
\[ + \Psi_t(s^t) u_c(s^t) z(s^t) - \gamma_t(s^t) u_c(s^t) b_t(s^{t-1}) \right\} \]

where

\[ \Psi_t(s^t) = \Psi_{t-1}(s^{t-1}) + \gamma_t(s^t) \quad \text{and} \quad \Psi_{-1}(s^{-1}) = 0 \quad (13) \]

In (12), the second equality uses the law of iterated expectations and Abel’s summation formula (also called \textit{summation by parts}, see this page).

First-order conditions with respect to \( c_t(s^t) \) can be expressed as

\[ u_c(s^t) - u_c(s^t) + \Psi_t(s^t) \{ [u_{cc}(s^t) - u_{cd}(s^t)] z(s^t) + u_c(s^t) z_c(s^t) \} \]
\[ - \gamma_t(s^t) [u_{cc}(s^t) - u_{cd}(s^t)] b_t(s^{t-1}) = 0 \quad (14) \]

and with respect to \( b_t(s^t) \) as

\[ \mathbb{E}_t [\gamma_{t+1}(s^{t+1}) u_c(s^{t+1})] = 0 \quad (15) \]

If we substitute \( z(s^t) \) from (7) and its derivative \( z_c(s^t) \) into first-order condition (14), 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 (14) does not appear in the corresponding expression for the Lucas-Stokey economy.

   - This term reflects the constraint that beginning-of-period government indebtedness must be the same across all realizations of next period’s state, a constraint that would not be present if government debt could be state contingent.

2. The Lagrange multiplier \( \Psi_t(s^t) \) in first-order condition (14) 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:

\[ \text{In [3]: using QuantEcon, NLsolve, NLopt} \]
\[ \text{import QuantEcon.simulate} \]
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 
    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 
    # 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 

func
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 = \(\text{LinearAlgebra.I} - β \cdot Π\), 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, II, G, Uc, Ucc, Un, Unn =
        model.Θ, model.β, model.II, 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
        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 = \(\text{LinearAlgebra.I} - β \cdot \text{model.II}, I\)
    return c, n, x, Ξ
end

function time0_allocation(pas::SequentialAllocation,
                          B_::AbstractFloat, s_0::Integer)
    model = pas.model
    II, Θ, G, β = model.II, 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[3], 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]
)

# 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...,)

function time1_value(pas::SequentialAllocation, μ::Real)
model = pas.model
(c, n, x, Ξ) = time1_allocation(pas, μ)
U_val = model.U(c, n)
V = \( \text{LinearAlgebra.I} - \text{model.β*model.Π} \), 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 .* Uc)
end

function simulate(pas::SequentialAllocation, B_::AbstractFloat, s_0::Integer,
                 T::Integer, sHist::Union{Vector, Nothing}=nothing)
model = pas.model
Π, β, Uc = model.Π, model.β, model.Uc
if isnothing(sHist)
  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, chHist[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 = U_c(c,n)
s = s Hist[t]
T Hist[t] = T(pas.model, c, n)[s]
E u_c = dot(II[s Hist[t - 1], :], u_c)
c Hist[t], n Hist[t], B Hist[t] = c[s], n[s], x[s] / u_c[s]
R Hist[t - 1] = U_c(c Hist[t - 1], n Hist[t - 1]) / (β * Eu_c)
μ Hist[t] = μ
end
return c Hist, n Hist, B Hist, T Hist, s Hist, μ Hist, RH Hist
end

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

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, 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, II = model.β, model.Θ, model.G, model.II
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(II[s, :], Vprime))
end

function cons(z::Vector, grad)
c, xprime = z[1], z[2:end]
n = c + G[s]
3. COMPETITIVE EQUILIBRIUM WITH DISTORTING TAXES

\[ \text{return } x - U_c(c, n) * c - U_n(c, n) * n - \beta * \text{dot}(\Pi[s, :], x') \]

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)
T.z0[i_x, s] = vcat(minx[1], minx[1] + G[s], minx[2:end])
return vcat(-minf, T.z0[i_x, s])
end

function get_policies_time0(T::BellmanEquation, B_::AbstractFloat, s0::Integer, Vf::Array)
    model, S = T.model, T.S
    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: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 -U(c, n) * (c - B_) - U_n(c, n) * n - β * dot(Π[s0, :], x')
    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)
return vcat(-minf, vcat(minx[1], minx[1]+G[s0], minx[2:end]))
end

Out[3]: get_policies_time0 (generic function with 1 method)

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.

63.4 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 (8) from the Lucas-Stokey economy, but
- adds measurability constraints (6) 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.

63.4.1 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 [2], 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 household’s first-order conditions are

\[
u_{c,t} = \beta R_t \mathbb{E}_tu_{c,t+1}
\]

Using these to eliminate \( R_t \) and \( \tau_t \) from budget constraint (16) gives

\[
b_t(s^{t-1}) + \frac{u_{c,t}(s^t)}{u_{c,t}(s^t)}n_t(s^t) = c_t(s^t) + \frac{\beta(\mathbb{E}_tu_{c,t+1})b_{t+1}(s^t)}{u_{c,t}(s^t)}
\]

or

\[
u_{c,t}(s^t)b_t(s^{t-1}) + u_{c,t}(s^t)n_t(s^t) = u_{c,t}(s^t)c_t(s^t) + \beta(\mathbb{E}_tu_{c,t+1})b_{t+1}(s^t)
\]
Now define
\[ x_t \equiv \beta b_{t+1}(s^t) \mathbb{E}_t u_{c,t+1} = u_{c,t}(s^t) \frac{b_{t+1}(s^t)}{R_t(s^t)} \]  \hspace{1cm} (19)
and represent the household’s budget constraint at time \( t \), history \( s^t \) as
\[ \frac{u_{c,t}x_{t-1}}{\beta \mathbb{E}_{t-1} u_{c,t}} = u_{c,t}c_t - u_{l,t}n_t + x_t \]  \hspace{1cm} (20)
for \( t \geq 1 \).

### 63.4.2 Measurability Constraints

Write equation (18) as
\[ b_t(s^{t-1}) = c_t(s^t) - \frac{u_{l,t}(s^t)}{u_{c,t}(s^t)} n_t(s^t) + \frac{\beta(\mathbb{E}_t u_{c,t+1})b_{t+1}(s^t)}{u_{c,t}} \]  \hspace{1cm} (21)

The right side of equation (21) 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 (21) must equal \( b_t(s^{t-1}) \).

That implies that it is has to be measurable with respect to \( s^{t-1} \).

Equations (21) are the measurability constraints that the AMSS model adds to the single time 0 implementation constraint imposed in the Lucas and Stokey model.

### 63.4.3 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
\[ V(x_-, s_-) = \max_{\{n(s), x(s)\}} \sum_s \Pi(s|s_-) [u(n(s) - g(s), 1 - n(s)) + \beta V(x(s), s)] \]  \hspace{1cm} (22)

subject to the following collection of implementability constraints, one for each \( s \in S \):
\[ \frac{u_c(s)x_-}{\beta \sum_{\tilde{s}} \Pi(\tilde{s}|s_-) u_{c}(\tilde{s})} = u_c(s)(n(s) - g(s)) - u_l(s)n(s) + x(s) \]  \hspace{1cm} (23)
A continuation Ramsey planner at $t \geq 1$ takes $(x_{t-1}, s_{t-1}) = (x_-, s_-)$ as given and before $s$ is realized chooses $(n_t(s_t), x_t(s_t)) = (n(s), x(s))$ for $s \in S$.

The Ramsey planner takes $(b_0, s_0)$ as given and chooses $(n_0, x_0)$. The value function $W(b_0, s_0)$ for the time $t = 0$ Ramsey planner satisfies the Bellman equation

$$W(b_0, s_0) = \max_{n_0, x_0} u(n_0 - g_0, 1 - n_0) + \beta V(x_0, s_0)$$

where maximization is subject to

$$u_{c,0} b_0 = u_{c,0}(n_0 - g_0) - u_{l,0} n_0 + x_0$$

### 63.4.4 Martingale Supercedes State-Variable Degeneracy

Let $\mu(s|s_-)\Pi(s|s_-)$ be a Lagrange multiplier on constraint (23) for state $s$.

After forming an appropriate Lagrangian, we find that the continuation Ramsey planner’s first-order condition with respect to $x(s)$ is

$$\beta V_x(x(s), s) = \mu(s|s_-)$$

Applying the envelope theorem to Bellman equation (22) gives

$$V_x(x_-, s_-) = \sum_s \Pi(s|s_-) \mu(s|s_-) \frac{u_c(s)}{\beta \sum_{\tilde{s}} \Pi(\tilde{s}|s_-) u_c(\tilde{s})}$$

Equations (26) and (27) imply that

$$V_x(x_-, s_-) = \sum_s \left( \Pi(s|s_-) \frac{u_c(s)}{\sum_{\tilde{s}} \Pi(\tilde{s}|s_-) u_c(\tilde{s})} \right) V_x(x(s), s)$$

Equation (28) 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:

$$\tilde{\Pi}(s|s_-) \equiv \Pi(s|s_-) \frac{u_c(s)}{\sum_{\tilde{s}} \Pi(\tilde{s}|s_-) u_c(\tilde{s})}$$

**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.

### 63.4.5 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.

### 63.4.6 Digression on Nonnegative Transfers

Throughout this lecture we have imposed that transfers $T_t = 0$.

AMSS [2] instead imposed a nonnegativity constraint $T_t \geq 0$ on transfers.

They also considered a special case of quasi-linear preferences, $u(c,l) = c + H(l)$.

In this case, $V_x(x,s) \leq 0$ is a non-positive martingale.

By the martingale convergence theorem $V_x(x,s)$ converges almost surely.

Furthermore, when the Markov chain $\Pi(s|s_{-})$ and the government expenditure function $g(s)$ are such that $g_t$ is perpetually random, $V_x(x,s)$ almost surely converges to zero.

For quasi-linear preferences, the first-order condition with respect to $n(s)$ becomes

$$(1 - \mu(s|s_{-}))(1 - u_I(s)) + \mu(s|s_{-})n(s)u_I(s) = 0$$

When $\mu(s|s_{-}) = \beta V_x(x(s),x)$ converges to zero, in the limit $u_I(s) = 1 = u_c(s)$, 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.

### 63.4.7 Code

The recursive formulation is implemented as follows

In [4]: using Dierckx

```julia
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}
```

---

---
struct RecursiveAllocation{TP <: Model, 
    TI <: Integer, 
    TVg <: AbstractVector, 
    TT <: Tuple}

    model::TP
    mc::MarkovChain
    S::TI
    T::BellmanEquationRecursive
    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_time1_bellman(model, μgrid)
    T.time_0 = true       # Bellman equation now solves time 0

    return RecursiveAllocation(model, mc, S, T, μgrid, xgrid, Vf, policies)
end

function solve_time1_bellman(model::Model{TR}, μgrid::AbstractArray) where {TR <: Real}
    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_:::AbstractFloat,
                                     s_:::Integer)
        c, n, x, V = time1_value(PP, μ_)
        return c, n, dot(Π[s_, :], x), dot(Π[s_, :], V)
    end

    cf = Array{Function}(undef, S, S)
    nf = Array{Function}(undef, S, S)
    xprimef = Array{Function}(undef, S, S)
    Vf = Vector{Function}(undef, S)
    xgrid = Array{TR}(undef, S, length(μgrid))

    for s_ in 1:S
        c = Array{TR}(undef, length(μgrid), S)
        n = Array{TR}(undef, length(μgrid), S)
        x = Array{TR}(undef, length(μgrid))
        V = Array{TR}(undef, length(μgrid))
for (i_μ, μ) in enumerate(μgrid)
    c[i_μ, :], n[i_μ, :], x[i_μ], V[i_μ] =
        incomplete_allocation(PP, μ, s_)
end
xprimes = repeat(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])
end
splV = Spline1D(x[end:-1:1], V[end:-1:1], k=3)
Vf[s_] = y -> splV(y)
# Vf[s_] = LinInterp(x[end:-1:1], V[end:-1:1])
end
policies = [cf, nf, xprimef]

# Create xgrid
xbar = [maximum(minimum(xgrid)), minimum(maximum(xgrid))]
xgrid = range(xbar[1], xbar[2], length = 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 = ", diff)
    Vf = copy(Vfnew)
end
return Vf, policies, T, xgrid
end

function fit_policy_function(T::BellmanEquation_Recursive,
                              PF::Function,
                              xgrid::AbstractVector{TF}) where {TF <: AbstractFloat}
    S = T.S
    # preallocation
    PFvec = Array{TF}(undef, 4S + 1, length(xgrid))
CHAPTER 63. OPTIMAL TAXATION WITHOUT STATE-CONTINGENT DEBT

\[
\begin{align*}
cf &= \text{Array\{Function\}}(\text{undef, S, S}) \\
nf &= \text{Array\{Function\}}(\text{undef, S, S}) \\
xprimef &= \text{Array\{Function\}}(\text{undef, S, S}) \\
TTf &= \text{Array\{Function\}}(\text{undef, S, S}) \\
Vf &= \text{Vector\{Function\}}(\text{undef, S}) \\
\end{align*}
\]

# fit policy functions

\[
\begin{align*}
\text{for } s_\text{in } 1:S \\
&\text{for } (i_\text{x}, x) \text{ in enumerate(xgrid)} \\
&\quad PFvec[s_, i_\text{x}] = PF(i_\text{x}, x, s_) \\
&\text{end} \\
&\text{splV} = \text{Spline1D}(xgrid, PFvec[1, :], k=3) \\
&Vf[s_] = y -> \text{splV}(y) \\
&\# Vf[s_] = \text{LinInterp}(xgrid, PFvec[1, :]) \\
&\text{for } sp\text{rime}=1:S \\
&\quad splc = \text{Spline1D}(xgrid, PFvec[1 + sp\text{rime}, :], k=3) \\
&\quad spln = \text{Spline1D}(xgrid, PFvec[1 + S + sp\text{rime}, :], k=3) \\
&\quad splxprime = \text{Spline1D}(xgrid, PFvec[1 + 2S + sp\text{rime}, :], k=3) \\
&\quad splTT = \text{Spline1D}(xgrid, PFvec[1 + 3S + sp\text{rime}, :], k=3) \\
&\quad cf[s_, sp\text{rime}] = y -> \text{splc}(y) \\
&\quad nf[s_, sp\text{rime}] = y -> \text{spln}(y) \\
&\quad xprimef[s_, sp\text{rime}] = y -> \text{splxprime}(y) \\
&\quad TTf[s_, sp\text{rime}] = y -> \text{splTT}(y) \\
&\text{end} \\
&\text{policies} = (cf, nf, xprimef, TTf) \\
&\text{return } Vf, \text{policies} \\
&\end{align*}
\]

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])

function time0_allocation(pab::RecursiveAllocation, B_::Real, s0::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

function simulate(pab::RecursiveAllocation,
                  B_::TF, s_0::Integer, T::Integer,
                  sHist::Vector=simulate(pab.mc, T, init=s_0)) where {TF <: AbstractFloat}
model, mc, Vf, S = pab.model, pab.mc, pab.Vf, pab.S
II, Uc = model.II, model.Uc
cf, nf, xprimef, TTf = pab.policies
63.4. RECURSIVE VERSION OF AMSS MODEL

```plaintext
chHist = Array(TF)(undef, T)
nHist = Array(TF)(undef, T)
Bhist = Array(TF)(undef, T)
xHist = Array(TF)(undef, T)
TauHist = Array(TF)(undef, T)
THist = Array(TF)(undef, T)
μHist = Array(TF)(undef, T)

#time 0
chHist[1], nHist[1], xHist[1], THist[1] = time0_allocation(pab, B_, 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)(undef, S)
  n = Array(TF)(undef, S)
  xprime = Array(TF)(undef, S)
  TT = Array(TF)(undef, S)
  for sprime=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(Π[s_, :], u_c)
  μHist[t] = Vf[s][xprime[s]]
  xHist[t], THist[t] = xprime[s], TT[s]
end
return chHist, nHist, Bhist, xHist, TauHist, THist, μHist, sHist
end
```

```plaintext
function BellmanEquation_Recursive(model::Model{TF},
xgrid::AbstractVector{TF},
policies0::Array) where {TF <: AbstractFloat}

S = size(model.Π, 1) # number of states
xbar = [minimum(xgrid), maximum(xgrid)]
time_0 = false
z0 = Array(Array)(undef, 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)(undef, S)
    ns = Array(TF)(undef, S)
    xprimes = Array(TF)(undef, S)
    for j = 1:S
      cs[j], ns[j], xprimes[j] = cf[s, j](x), nf[s, j](x),
    end
  end
end
return chHist, nHist, Bhist, xHist, TauHist, THist, μHist, sHist
end
```

```plaintext
S = size(model.Π, 1) # number of states
xbar = [minimum(xgrid), maximum(xgrid)]
time_0 = false
z0 = Array(Array)(undef, 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)(undef, S)
    ns = Array(TF)(undef, S)
    xprimes = Array(TF)(undef, S)
    for j = 1:S
      cs[j], ns[j], xprimes[j] = cf[s, j](x), nf[s, j](x),
    end
  end
end
return chHist, nHist, Bhist, xHist, TauHist, THist, μHist, sHist
end
```
end
    z0[i_x, s] = vcat(cs, ns, xprimes, zeros(S))
end

end

end

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, II = model.β, model.Θ, model.G, model.II
    Uc, Un = model.Uc, model.Un

    S_possible = sum(II[s_, :]>0)
    sprimei_possible = findAll(II[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(II[s_, sprimei_possible], Uc(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(II[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(II[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
        lb = vcat(zeros(S_possible), ones(S_possible)*xbar[1], zeros(S_possible))
    end

    if model.n_less_than_one == true
        ub = vcat(ones(S_possible) - G[sprimei_possible], ones(S_possible) * xbar[2], ones(S_possible))
    else
        ub = vcat(100 * ones(S_possible), ones(S_possible) * xbar[2], zeros(S_possible))
    end

    lb, ub, opt = find_first_best(model, S, 2)
end

function BellmanEquation_Recursive(model, S, xbar, time_0, z0, cFB, nFB, IFB, xFB, zFB)

    return lb, ub, opt
end

if model.transfers == true
    lb = vcat(zeros(S_possible), ones(S_possible)*xbar[1], zeros(S_possible))
end

if model.n_less_than_one == true
    ub = vcat(ones(S_possible) - G[sprimei_possible], ones(S_possible) * xbar[2], ones(S_possible))
else
    ub = vcat(100 * ones(S_possible), ones(S_possible) * xbar[2], zeros(S_possible))
end
100 * ones(S_possible))

end
init = vcat(T.z0[i_x, s_][sprimei_possible],
T.z0[i_x, s_][2S + sprimei_possible],
T.z0[i_x, s_][3S + sprimei_possible])

opt = Opt(:LN_COBYLA, 3S_possible)

equality_constraint!(opt, cons, zeros(S_possible))

else
lb = vcat(zeros(S_possible), ones(S_possible)*xbar[1])

if model.n_less_than_one == true
ub = vcat(ones(S_possible) - G[sprimei_possible],
ones(S_possible)*xbar[2])
else
ub = vcat(ones(S_possible), ones(S_possible) * xbar[2])
end

init = vcat(T.z0[i_x, s_][sprimei_possible],
T.z0[i_x, s_][2S + sprimei_possible])

opt = Opt(:LN_COBYLA, 2S_possible)

equality_constraint!(opt, cons_no_trans, zeros(S_possible))

end

init[init .> ub] = ub[init .> ub]

min_objective!(opt, objf)
lower_bounds!(opt, lb)
upper_bounds!(opt, ub)

maxeval!(opt, 10000000)
maxtime!(opt, 10)
ftol_rel!(opt, 1e-8)
ftol_abs!(opt, 1e-8)

(minf, minx, ret) = optimize(opt, init)

if ret != :SUCCESS && ret != :ROUNDOFF_LIMITED && ret != :
MAXEVAL_REACHED &&
ret != :FTOL_REACHED && ret != :MAXTIME_REACHED
error("optimization failed: ret = $ret")

end

T.z0[i_x, s_][sprimei_possible] = minx[1:S_possible]
T.z0[i_x, s_][S + sprimei_possible] = minx[1:S_possible] + 0

G[sprimei_possible]
T.z0[i_x, s_][2S + sprimei_possible] = minx[2S_possible + 1:
2S_possible]

if model.transfers == true
T.z0[i_x, s_][3S + sprimei_possible] = minx[2S_possible + 1:
3S_possible]
else
T.z0[i_x, s_][3S + sprimei_possible] = zeros(S)
end

end

return vcat(-minf, T.z0[i_x, s_])

function get_policies_time0(T::BellmanEquation_Recursive,
B::Real,
s0::Integer,
CHAPTER 63. OPTIMAL TAXATION WITHOUT STATE-CONTINGENT DEBT

\[ Vf : \text{AbstractArray}\{\text{Function}\}, \]
\[ xbar : \text{AbstractVector} \]

model = T.model
\( \beta, \Theta, G = \text{model.}\beta, \text{model.}\Theta, \text{model.G} \)
U, Uc, Un = model.U, model.Uc, model.Un

function objf(z, grad)
c, xprime = z[1], z[2]
n = (c + G[s0]) / \Theta[s0]
\[ \text{return } -U(c, n) + \beta \ast Vf[s0](xprime) \]
end

function cons(z, grad)
c, xprime, TT = z[1], z[2], z[3]
n = (c + G[s0]) / \Theta[s0]
\[ \text{return } -Uc(c, n) \ast (c - B_ - TT) - Un(c, n) \ast n - \beta \ast xprime \]
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)
equality_constraint!(opt, cons_no_trans)
end
init[init .> ub] = ub[init .> ub]
init[init .< lb] = lb[init .< lb]

min_objective!(opt, objf)
lower_bounds!(opt, lb)
upper_bounds!(opt, ub)
maxeval!(opt, 100000000)
maxtime!(opt, 30)

(minf, minx, ret) = optimize(opt, init)

if ret != :SUCCESS && ret != :ROUNDOFFLIMITED && ret != :
  MAXEVAL_REACHED &&
  ret != :FTOL_REACHED
  error("optimization failed: ret = $ret")

end

if model.transfers == true
    return -minf, minx[1], minx[1]+G[0], minx[2], minx[3]
else
    return -minf, minx[1], minx[1]+G[0], minx[2], 0
end

```
Out[4]: get_policies_time0 (generic function with 2 methods)
```

### 63.5 Examples

We now turn to some examples.

#### 63.5.1 Anticipated One-Period War

In our lecture on optimal taxation with state contingent debt we studied how the government manages uncertainty in a simple setting. As in that lecture, we assume the one-period utility function

\[
    u(c, n) = \frac{c^{1-\sigma}}{1-\sigma} - \frac{n^{1+\gamma}}{1+\gamma}
\]

**Note**

For convenience in matching our computer code, we have expressed utility as a function of \( n \) rather than leisure \( l \)

We consider the same government expenditure process studied in the lecture on optimal taxation with state contingent debt.

Government expenditures are known for sure in all periods except one

- For \( t < 3 \) or \( t > 3 \) we assume that \( g_t = g_l = 0.1 \).
- At \( t = 3 \) a war occurs with probability 0.5.
  - If there is war, \( g_3 = g_h = 0.2 \).
  - If there is no war \( g_3 = g_l = 0.1 \).

A useful trick is to define components of the state vector as the following six \((t, g)\) pairs:

\[
    (0, g_l), (1, g_l), (2, g_l), (3, g_l), (3, g_h), (t \geq 4, g_l)
\]

We think of these 6 states as corresponding to \( s = 1, 2, 3, 4, 5, 6 \).

The transition matrix is
\[
P = \begin{pmatrix}
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0.5 & 0.5 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}
\]

The government expenditure at each state is
\[
g = \begin{pmatrix}
0.1 \\
0.1 \\
0.1 \\
0.1 \\
0.2 \\
0.1
\end{pmatrix}
\]

We assume the same utility parameters as in the Lucas-Stokey economy.

This utility function is implemented in the following constructor

In [5]: 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),
\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.^(\gamma) / (1 + \gamma)
end

# Derivatives of utility function
Uc(c,n) = c.^(-\sigma)
Ucc(c,n) = -\sigma \times c.^(-\sigma - 1.0)
Un(c,n) = -n.^\gamma
Unn(c,n) = -\gamma \times n.^{\gamma - 1.0}
n_less_than_one = \text{false}
return Model(\beta, \Pi, G, \Theta, \text{transfers},
            U, Uc, Ucc, Un, Unn, n_less_than_one)
end

Out[5]: crra_utility (generic function with 1 method)

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.

In [6]: time_example = crra_utility(G=[0.1, 0.1, 0.1, 0.2, 0.1, 0.1],
          Θ = ones(6))  # Θ can in principle be random

time_example.Π = [
    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;
    0.0 0.0 0.0 0.0 0.0 1.0]

# Initialize μgrid for value function iteration
μgrid = range(-0.7, 0.01, length = 200)

time_example.transfers = true  # Government can use transfers
time_sequential = SequentialAllocation(time_example)  # Solve sequential problem

time_bellman = RecursiveAllocation(time_example, μgrid)

sHist_h = [1, 2, 3, 4, 6, 6]
sHist_l = [1, 2, 3, 5, 6, 6]

sim_seq_h = simulate(time_sequential, 1., 1, 7, sHist_h)
sim_bel_h = simulate(time_bellman, 1., 1, 7, sHist_h)
sim_seq_l = simulate(time_sequential, 1., 1, 7, sHist_l)
sim_bel_l = simulate(time_bellman, 1., 1, 7, sHist_l)

using Plots
gr(fmt=:png);
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],
        time_example.Θ[sHist_l] .* sim_bel_l[2])
sim_seq_h_plot = hcat(sim_seq_h[1:3], ..., sim_seq_h[4],
        time_example.G[sHist_h],
        time_example.Θ[sHist_h] .* sim_seq_h[2])
sim_bel_h_plot = hcat(sim_bel_h[1:3], ..., sim_bel_h[5],
        time_example.G[sHist_h],
        time_example.Θ[sHist_h] .* sim_bel_h[2])

p = plot(size = (920, 750), layout = (3, 2),
xaxis=(0:6), grid=false, titlefont=Plots.font("sans-serif", 10))
plot!(p, title = titles)
for i=1:6
    plot!(p[i, 0:6, sim_seq_l_plot[:, i], marker=:circle, color=:black,
          lab="")
    plot!(p[i, 0:6, sim_bel_l_plot[:, i], marker=:circle, color=:red,
          lab="")
    plot!(p[i, 0:6, sim_seq_h_plot[:, i], marker=:utriangle, color=:black, lab="")
    plot!(p[i, 0:6, sim_bel_h_plot[:, i], marker=:utriangle, color=:red, lab="")
end
\texttt{diff = 0.01954181139136142}
\texttt{diff = 0.020296606343314445}
\texttt{diff = 0.01825290333882874}
\texttt{diff = 0.01732330130265356}
\texttt{diff = 0.002438290769982288}
\texttt{diff = 0.0017049947758874273}
\texttt{diff = 0.001491200081454862}
\texttt{diff = 0.000841571606329687}
\texttt{diff = 0.000344132585210494}
\texttt{diff = 0.000300898305458036}
\texttt{diff = 0.0002797536359176724}
\texttt{diff = 0.0002436270260934893}
\texttt{diff = 0.0002192208026612293}
\texttt{diff = 0.00019723701023064875}
\texttt{diff = 0.000177505553015148367}
\texttt{diff = 0.00015972217622243714}
\texttt{diff = 0.00014373496579809245}
\texttt{diff = 0.00012934400797986716}
\texttt{diff = 0.00011639475826865351}
\texttt{diff = 0.00010474335675915266}
\texttt{diff = 9.4259404048251e-5}
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

```
In [7]: 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)
    U(c,n) = log(c) + ψ * log(1 - n)
    # Derivatives of utility function
    Uc(c,n) = 1 ./ c
```
Ucc(c,n) = -c.^(2.0)
Un(c,n) = -ψ ./(1.0 .- n)
Unn(c,n) = -ψ ./(1.0 .- n).^2.0
n_less_than_one = true
return Model(β, Π, G, Θ, transfers, U, Uc, Ucc, Un, Unn, n_less_than_one)
end

Out[7]: log_utility (generic function with 1 method)

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).

In [8]: 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, 1, 1, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1]

#simulate
sim_seq = simulate(log_sequential, 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 = simulate(log_bellman, 0.5, 1, T, sHist)
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 = (920, 750), layout = grid(3, 2),
    xaxis=(0:T), grid=false, titlefont=Plots.font("sans-serif", 10))
labels = fill(("", ""), 6)
labels[3] = ("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=:black,
    lab=labels[i][1])
    plot!(p[i], sim_bel_plot[:, i], marker=:utriangle, color=:blue,
    lab=labels[i][2],
    legend=:bottomright)
end
plot!(p[5], sim_seq_plot[:, 5], marker=:circle, color=:blue, lab="")

diff = 0.0007972378476372139
diff = 0.0006423560333504441
diff = 0.0005517441622530832
diff = 0.00048553930013351857
diff = 0.0004226590836939342
diff = 0.00037550672316976404
diff = 0.0003294032122270672
diff = 0.00029337232321791915
diff = 0.00025856795048240623
diff = 0.00023042624865279873
diff = 0.0002031214087191915
diff = 0.00018115282833643646
diff = 0.00016034374751970243
diff = 0.00014294960573402432
diff = 0.00012675817151896933
diff = 0.00011295205489914281
diff = 0.00010390878977062024
diff = 8.934103186095062e-5

Out[8]:

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.
In [9]: `T_long = 200`
   `sim_seq_long = simulate(log_sequential, 0.5, 1, T_long)`
   `sHist_long = sim_seq_long[end-2]`
   `sim_bel_long = simulate(log_bellman, 0.5, 1, T_long, sHist_long)`
   `sim_seq_long_plot = hcat(sim_seq_long[1:4]...,
     log_example.G[sHist_long], log_example.Θ[sHist_long] .*` sim_seq_long[2])
   `sim_bel_long_plot = hcat(sim_bel_long[1:3]...,
     sim_bel_long[5],
     log_example.G[sHist_long], log_example.Θ[sHist_long] .*` sim_bel_long[2])
   `p = plot(size = (920, 750), layout = (3, 2), xaxis=(0:50:T_long),
     grid=false,
     titlefont=Plots.font("sans-serif", 10))`
   `plot!(p, title = titles)`
   `for i = 1:6
     plot!(p[i], sim_seq_long_plot[:, i], color=:black, linestyle=:solid,
       lab=labels[i][1])
     plot!(p[i], sim_bel_long_plot[:, i], color=:blue, linestyle=:dot,
       lab=labels[i][2],
       legend=:bottomright)`
   `end
   p`

Out[9]:

Footnotes
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.

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 (11) takes a positive value, so \( \gamma_t(\tilde{s}^t) > 0 \).
Bibliography


