Quantitative Economics with Julia

JESSE PERLA, THOMAS J. SARGENT AND JOHN STACHURSKI

February 19, 2020
Contents
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.
CHAPTER 1. SETTING UP YOUR JULIA ENVIRONMENT

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-1.3.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 = "0.5.0", instantiate = true)
1.4. DESKTOP INSTALLATION OF JULIA AND JUPYTER

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/) then the add IJulia installs everything you need into your existing environment.

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

```julia
using IJulia; jupyterlab()
```

Alternatively, if you installed Jupyter separately in Jupyter Installation or added Jupyter to your path (#add-jupyter-to-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 contact@quantecon.org for help on setting up a shared JupyterHub instance with precompiled packages ready for these lecture notes.
- JuliaBox tightly controls allowed packages, and does not currently support the Quantecon lectures.

**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>`) [1]:

```jupyter
] add InstantiateFromURL
```

Then, run

```jupyter
using InstantiateFromURL
github_project("QuantEcon/quantecon-notebooks-julia", version = "0.5.0", instantiate = 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 1. SETTING UP YOUR JULIA ENVIRONMENT
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

```
[1]: using InstantiateFromURL
github_project("QuantEcon/quantecon-notebooks-julia", version = "0.5.0")
# github_project("QuantEcon/quantecon-notebooks-julia", version = "0.5.0", instantiate =false)
```

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

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
2.3. USING JUPYTER

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`.

![Tab completion example in Jupyter](image)

**Getting Help**

To get help on the Julia function such as `repeat`, enter `? repeat`. Documentation should now appear in the browser.

```
In [1]: ? repeat
search: repeat
Out[1]:
```

**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.

```
Euler found that
$$\exp(i \pi) = -1$$
```

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.
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, \(; \text{ls}\) will execute the UNIX style shell command \text{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.

Package Operations

You can execute package operations in the notebook by prepending a \].

For example, \(] \text{st}\) 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.
2.5 (Optional) Adding Jupyter to the Path

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 \texttt{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 \texttt{.julia} folder (e.g., \texttt{"C:\Users\USERNAME\.julia\conda\3\Scripts} on Windows).

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

\begin{verbatim}
] add Conda
using Conda
Conda.SCRIPTDIR
\end{verbatim}

On Linux/OSX, you could add that path to your \texttt{.bashrc}.

On Windows, to add directly to the path, type \texttt{;} to enter shell mode and then execute

\begin{verbatim}
setx PATH "$\{Conda.SCRIPTDIR\};%PATH%"
\end{verbatim}
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

[1]:

```julia
using InstantiateFromURL
github_project("QuantEcon/quantecon-notebooks-julia", version = "0.5.0") # github_project("QuantEcon/quantecon-notebooks-julia", version = "0.5.0", instantiate =false) # uncomment to force package installation
```

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.
3.3. EXAMPLE: PLOTTING A WHITE NOISE PROCESS

This code will download and install project files from GitHub, QuantEcon/QuantEconLecturePackages.

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.

\[2\]: 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.

\[3\]:
\[
\text{randn()}
\]
\[3\]: 0.669334635697782

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

\[4\]:
\[
\text{using Plots}
\]
\[
\text{gr(fmt=:png); # setting for easier display in jupyter notebooks}
\]
\[
n = 100
\]
\[
\vec{\omega} = \text{randn}(n)
\]
\[
\text{plot}(1:n, \vec{\omega})
\]
\[4\]:
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 ₋ 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 ₋ 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

[5]: typeof(ᵦ )

[5]: Array{Float64,1}
3.3. EXAMPLE: PLOTTING A WHITE NOISE PROCESS

The information from `typeof()` tells us that  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  returns an array of the first 5 elements of .

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.

?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
**CHAPTER 3. INTRODUCTORY EXAMPLES**

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

[7]:
```plaintext
# poor style
n = 100
ū = zeros(n)
for i in 1:n
    ē[i] = randn()
end
```

Here we first declared `ū` 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 `ē` or `ū =`. 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 `ū` with the correct values, it is very indirect as the connection between the index `i` and the `ū` vector is unclear.

To fix this, use **eachindex**

[8]:
```plaintext
# better style
n = 100
ū = zeros(n)
for i in eachindex(ū)
    ē[i] = randn()
end
```

Here, `eachindex(ū)` returns an iterator of indices which can be used to access `ū`. 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

[9]:
```plaintext
# careful to use 0.0 here, instead of 0
m = 5
for ē_val in ē[1:m]
    ē_sum = ē_sum + ē_val
end
ē_mean = ē_sum / m
```

[9]:
```
-0.41231379732739304
```

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

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

### 3.3.5 User-Defined Functions

For the sake of the exercise, let’s go back to the \texttt{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.

```plaintext
# poor style
function generatedata(n)
    v = zeros(n)
    for i in eachindex(v)
        v[i] = (randn())^2 # squaring the result
    end
    return v
end

data = generatedata(10)
plot(data)
```

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.

```julia
# still poor style
function generatedata(n)
    x = randn(n) # use built in function
    for i in eachindex(x)
        x[i] = x[i]^2 # squaring the result
    end
    return x
end
data = generatedata(5)
```

```
5-element Array{Float64,1}:
0.3611304498553976
0.31546691367490387
1.176863288499433
0.8020612841889548
3.149967158908303
```

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.

```julia
# better style
function generatedata(n)
    x = randn(n) # use built in function
    return x.^2
end
data = generatedata(5)
```

```
5-element Array{Float64,1}:
1.155389645151567
0.6803471018042372
0.8230831598180738
0.8523073779919349
1.8964630159068732
```

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

```julia
# good style
generatedata(n) = randn(n).^2
data = generatedata(5)
```

```
5-element Array{Float64,1}:
0.7787191126977215
0.3023700617062767
0.7205564280440987
0.021094246775535938
0.6648013091635033
```
Finally, we can broadcast any function, where squaring is only a special case.

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

As a final – abstract – approach, we can make the `generatedata` function able to generically apply to a function.

```
generatedata(n, gen) = gen.(randn(n))  # uses broadcast for some function `gen`
f(x) = x^2  # simple square function
data = generatedata(5, f)  # applies f
```

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.

```
# 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
```
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

```
[18]:
using Distributions

function plothistogram(distribution, n)
    # n draws from distribution
    histogram(rand(distribution, n))
end

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

```
[18]:
```
Let’s have a casual discussion of how all this works while leaving technical details for later in the lectures.

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

The name \( \texttt{lp} \) 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 \( \texttt{lp} \)
- the name \( \texttt{n} \) bound to the integer \( \texttt{500} \)

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 \([0,1)\).

\begin{verbatim}
[19]: rand(3)

3-element Array{Float64,1}:
  0.8805786430442772
  0.877826175516391
  0.5056162037738818
\end{verbatim}

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 \textbf{multiple dispatch}, which Julia uses to implement \textbf{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 \textbf{methods}.

\subsection*{3.4 Example: Variations on Fixed Points}

Take a mapping $f : X \rightarrow X$ 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 \textbf{Julian} style with high mathematical clarity.

\subsubsection*{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 $f(x) : \mathbb{R} \rightarrow \mathbb{R}$

$$ v = f(v) \quad (1) $$

where

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

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

\subsubsection*{3.4.2 While Loops}

One approach to finding a fixed point of Eq. (1) is to start with an initial value, and iterate the map
3.4. EXAMPLE: VARIATIONS ON FIXED POINTS

\[ v^{n+1} = f(v^n) \]  

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

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

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

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

```plaintext
[20]:
# poor style
p = 1.0 # note 1.0 rather than 1
beta = 0.9
maxiter = 1000
tolerance = 1.0E-7
v_iv = 0.8 # initial condition

# setup the algorithm
v_old = v_iv
normdiff = Inf
iter = 1
while normdiff > tolerance && iter <= maxiter
    v_new = p + beta * 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 = 

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

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

Here, we have used the \texttt{norm} function (from the \texttt{LinearAlgebra} base library) to compare the values.

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

An alternative approach is to use a \texttt{for} loop, and check for convergence in each iteration.

```
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.

```julia
# better, but still poor style
function v_fp(β, p, 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 \( p + \beta \cdot v \) is specific to our problem.

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

```julia
# better style
function fixedpointmap(f, iv, tolerance, maxiter)
    # setup the algorithm
    x_old = iv
    normdiff = Inf
```
3.4. EXAMPLE: VARIATIONS ON FIXED POINTS

```plaintext
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

# 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 = 

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

```plaintext
# good style
function fixedpointmap(f; iv, tolerance=1E-7, maxiter=1000)
    # 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 (value = x_old, normdiff, iter) # A named tuple
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!

sol = fixedpointmap(f, iv=0.8, tolerance=1.0E-8) # don't need to pass
println("Fixed point = 

Fixed point = 9.99999918629035, and |f(x) - x| = 9.841219328764782e-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 \( \text{iv} \) is required while \( \text{tolerance} \) and \( \text{maxiter} \) have default values.

The return type of the function also has named fields, \( \text{value} \), \( \text{normdiff} \), and \( \text{iter} \) – 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 \( f(x) := rx(1-x) \).

```plaintext
\[
[r = 2.0; f(x) = r \times x \times (1 - x)]
\]

\[
sol = \text{fixedpoint}\text{map}(f, \text{iv}=0.8)
\]

\[
\text{println("Fixed point = \$}(sol.\text{value}), \text{ and } |f(x) - x| = \$}(sol.\text{normdiff}) \text{ in } \$}(sol.\text{iter})\]
\]

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

### 3.4.6 Using a Package

But best of all is to avoid writing code altogether.

```plaintext
\[
# best style
\]

```plaintext
\[
\text{using NLsolve}
\]

```plaintext
\[
p = 1.0
\]

```plaintext
\[
\beta = 0.9
\]

```plaintext
\[
f(v) = p .+ \beta \times v \# broadcast the +
\]

```plaintext
\[
sol = \text{fixedpoint}(f, [0.8])
\]

```plaintext
\[
\text{println("Fixed point = \$}(sol.\text{zero}), \text{ and } |f(x) - x| = \$}(\text{norm}(f(sol.\text{zero}) - sol.\text{zero})) \text{ in } \$}(sol.\text{iterations}) \text{ iterations")}
\]

Fixed point = [9.999999999999972], and \(|f(x) - x| = 3.552713678800501e-15 \text{ in } 3 \text{ iterations}\)

The \text{fixedpoint} function from the \text{NLsolve.jl} library implements the simple fixed point iteration scheme above.

Since the \text{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.

```plaintext
\[
# best style
\]

```plaintext
\[
p = 1.0
\]

```plaintext
\[
\beta = 0.9
\]

```plaintext
\[
\text{iv} = [0.8]
\]

```plaintext
\[
sol = \text{fixedpoint}(v \rightarrow p .+ \beta \times v, \text{iv})
\]

```plaintext
\[
\text{println("Fixed point = \$}(sol.\text{zero}), \text{ and } |f(x) - x| = \$}(\text{norm}(f(sol.\text{zero}) - sol.\text{zero})) \text{ in } \$}(sol.\text{iterations}) \text{ iterations")}
\]

Fixed point = [9.999999999999972], and \(|f(x) - x| = 3.552713678800501e-15 \text{ in } 3 \text{ iterations}\)
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 Eq. (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 \( v \rightarrow p .+ \beta \cdot v \) 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

\[
\text{eps}() = 2.220446049250313e-16
\]

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, **BigFloat**, for the initial value.

```
# use arbitrary precision floating points
p = 1.0
β = 0.9
iv = [BigFloat(0.8)] # higher precision

# otherwise identical
sol = fixedpoint(v -> p .+ β * v, iv)
println("Fixed point = BigFloat[10.0000000000000002220446049250313573885329...3772 8786780993681155868155], 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:

```plaintext
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 = fixedpointmap(f, iv = iv, tolerance = 1.0E-8)
println("Fixed point = \$(sol.value), and |f(x) - x| = \$(sol.normdiff) in \$(sol.iter) iterations")
```

Fixed point = [9.999999961080519, 19.999999923853192], and |f(x) - x| = 9.501826248250528e-9 in 184 iterations

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

```plaintext
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 = 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 = [9.999999999999998, 20.0, 1.0], and |f(x) - x| = 0.0 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.

```plaintext
using NLsolve, StaticArrays
p = @SVector [1.0, 2.0, 0.1]
β = 0.9
iv = @SVector [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 = [9.999999999999998, 20.0, 1.0], and |f(x) - x| = 0.0 in 3 iterations
The \texttt{@SVector} in front of the \texttt{[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 \texttt{@} 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 \texttt{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 \texttt{true} with probability \( p \).

#### 3.5.3 Exercise 3

Compute an approximation to \( \pi \) using Monte Carlo.

For random number generation use only \texttt{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 = \( \pi \cdot radius^2 \).

#### 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 \texttt{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 \( x_0 = 1 \)

\[ x_{t+1} = \alpha x_t + \sigma \epsilon_{t+1} \quad \text{where} \quad x_0 = 1 \quad \text{and} \quad t = 0, \ldots, t_{\text{max}} \]

• 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 \textbf{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_prime`, `x_0`, `tolerance`, `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_prime` is calculated with auto-differentiation.

```plaintext
using ForwardDiff

# operator to get the derivative of this function using AD
D(f) = x -> ForwardDiff.derivative(f, x)

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

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

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 `ForwardDiff.jl`.

### 3.6 Solutions

#### 3.6.1 Exercise 1

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

factorial2(4)
```

```plaintext
24
```
3.6.2 Exercise 2

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

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

7, 4, 8, 4, 4, 5, 3, 3, 6, 6, 2, 5, 7, 4, 6, 3, 6, 6, 6, 3, 8, 7, 5, 0,

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.

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

3.139772
3.6.4 Exercise 4

```python
[38]:
    payoff = 0
    count = 0
    print("Count = ")
    for i in 1:10
        U = rand()
        if U < 0.5
            count += 1
        else
            count = 0
        print(count)
        if count == 3
            payoff = 1
        end
    end
    println("\n\npayoff = ")
```

Count = 0120000123
payoff = 1

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

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

[39]:
    "foo"

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

[40]:
    "bar"
```

Using this construction:

```python
[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("\n\npayoff = ")
```

Count = 0101234501
payoff = 1

3.6.5 Exercise 5

Here's one solution
40

CHAPTER 3. INTRODUCTORY EXAMPLES

3.6.6 Exercise 6

\[ \text{using Plots} \]
\[ \text{gr(fmt=:png); \ # setting for easier display in jupyter notebooks} \]
\[ \alpha = 0.9 \]
\[ n = 200 \]
\[ x = \text{zeros}(n + 1) \]

\[ \text{for \ } t \ \text{in} \ 1:n \]
\[ x[t+1] = \alpha \cdot x[t] + \text{randn()} \]
\[ \text{end} \]
\[ \text{plot}(x) \]

\[ \text{as} = [0.0, 0.8, 0.98] \]
\[ n = 200 \]
\[ p = \text{plot}(); \ # naming a plot to add to \]

\[ \text{for \ } \alpha \ \text{in} \ \text{as} \]
\[ x = \text{zeros}(n + 1) \]
\[ x[1] = 0.0 \]
\[ \text{for \ } t \ \text{in} \ 1:n \]
\[ x[t+1] = \alpha \cdot x[t] + \text{randn()} \]
\[ \text{end} \]
\[ \text{plot!}(p, x, \text{label} = "\alpha = \alpha") \ # add to plot \]
\[ \text{end} \]
\[ p \ # display plot \]
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

```python
function drawsuntilthreshold(threshold; maxdraws=100)
    for i in 1:maxdraws
        val = rand()
        if val < threshold # checks threshold
            return i # leaves function, returning draw number
        end
    end
    return Inf # if here, reached maxdraws
end

draws = drawsuntilthreshold(0.2, maxdraws=100)
```

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

```python
vals = zeros(0) # empty vector
for i in 1:100
    val = rand()
    if val < 0.5
        push!(vals, val)
    end
end
println("There were $(length(vals)) below 0.5")
```

There were 61 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

[1]: 
```
using InstantiateFromURL
github_project("QuantEcon/quantecon-notebooks-julia", version = "0.5.0")
# github_project("QuantEcon/quantecon-notebooks-julia", version = "0.5.0", instantiate = true) # uncomment to force package installation
```

[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 `true` or `false`.

[3]: 
```
x = true
```

[3]: 
```
true
```

[4]: 
```
typeof(x)
```

[4]: 
```
Bool
```

[5]: 
```
y = 1 > 2  # now y = false
```

[5]: 
```
false
```

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)

[6]: 
```
typeof(1.0)
```

[6]: 
```
Float64
```
4.3. COMMON DATA TYPES

[7]: typeof(1)

[7]: Int64

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

Arithmetic operations are fairly standard.

[8]: x = 2; y = 1.0;

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.

[9]: x * y

[9]: 2.0

[10]: x^2

[10]: 4

[11]: y / x

[11]: 0.5

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

[12]: 2x - 3y

[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.

[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.

[14]: x = 1 + 2im
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).

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

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

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

The ‘dollar’ inside of a string is used to interpolate a variable.

```julia
"x = \$x"
```

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

```julia
"x + y = \$(x + y)"
```

To concatenate strings use *

```julia
"foo" * "bar"
```
4.3. COMMON DATA TYPES

Julia provides many functions for working with strings.

```plaintext
[23]: s = "Charlie don't surf"
[24]: split(s)
[25]: replace(s, "surf" => "ski")
[26]: split("fee,fi,fo", ",")
[27]: strip(" foobar ") # remove whitespace
```

```plaintext
[22]: "foobar"
```

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

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

### 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.

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

```plaintext
[29]: ("foo", 2)
```
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.

```julia
x = "foo", 1

("foo", 1)
```

Tuples can also be unpacked directly into variables.

```julia
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.

```julia
x = ("foo", 1)
y = ("foo",)
typeof(x), typeof(y)
```

```julia
(Tuple{String,Int64}, Tuple{String})
```

### Referencing Items

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

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

```julia
4-element Array{Int64,1}:
10
20
30
40
```
4.3. COMMON DATA TYPES

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

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

The same slice notation works on strings.

```plaintext
[41]: "foobar"[3:end]
"obar"
```

**Dictionaries**

Another container type worth mentioning is dictionaries.

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

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

```plaintext
[43]: d["age"]
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.

```
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

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

123

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

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

Dict[String,Any] with 2 entries:
  "name" => "Frodo"
  "age" => 33

```
keys(d)
```

Base.KeySet for a Dict[String,Any] with 2 entries. Keys:
  "name"
  "age"
4.4. ITERATING

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()`.

```plaintext
[48]: collect(keys(d))
```

```plaintext
[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

```plaintext
[49]: x_values = 1:5
```

```plaintext
[49]: 1:5
```

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

```
1
4
9
16
25
```

```plaintext
[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
The capital of Japan is Tokyo
The capital of Korea is Seoul
The capital of China is Beijing

If we happen to need the index as well as the value, one option is to use `enumerate()`. The following snippet will give you the idea.

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

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

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

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

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

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

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

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

3×3 Array{Int64,2}:
5 6 7
6 7 8
[58]:
    [ i + j + k for i in 1:3, j in 4:6, k in 7:9 ]

[58]:
3×3×3 Array{Int64,3}:
[:, :, 1] =
12 13 14
13 14 15
14 15 16
[:, :, 2] =
13 14 15
14 15 16
15 16 17
[:, :, 3] =
14 15 16
15 16 17
16 17 18

Comprehensions can also create arrays of tuples or named tuples

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

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

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

[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")

4.5 Comparisons and Logical Operators

4.5.1 Comparisons

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

[61]:
x = 1

[61]:
1

[62]:
x == 2

[62]:
false

For “not equal” use != or ≠ (\ne<TAB>).

[63]:
x != 3
Julia can also test approximate equality with \( \approx \) (\approx\). 

\[
1 + 10^{-8} \approx 1
\]

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)

true && false

false

ture || false

true

Remember

- \( P \land Q \) is true if both are true, otherwise it’s false.
- \( P \lor Q \) 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. USER-DEFINED FUNCTIONS

4.6.1 Return Statement

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

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

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

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

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

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

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

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

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`:

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

Let’s check that it works:

```julia
f(1 / pi)
```

Julia also allows you to define anonymous functions.

For example, to define \( f(x) = \sin\left(\frac{1}{x}\right) \) you can use \( x \rightarrow \sin\left(\frac{1}{x}\right) \).
The difference is that the second function has no name bound to it.

How can you use a function with no name?

Typically it’s as an argument to another function

```
map(x -> sin(1 / x), randn(3))  # apply function to each element
```

```
3-element Array{Float64,1}:
  -0.7230725331797222
  -0.7542630853261694
  0.9967879394982074
```

### 4.6.3 Optional and Keyword Arguments

(See [keyword arguments documentation](https://docs.julialang.org/en/v1/manual/types/arguments/))

Function arguments can be given default values

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

```
f (generic function with 3 methods)
```

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

```
f(pi)
```

```
0.36787944117144233
```

```
f(pi, 2)
```

```
2.718281828459045
```

Another option is to use **keyword** arguments.

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

For example, in the call

```
f(x; a = 1) = exp(cos(a * x))  # note the ; in the definition
f(pi, a = 2)  # calling with ; is usually optional and generally discouraged
```

```
2.718281828459045
```

### 4.7 Broadcasting

(See [broadcasting documentation](https://docs.julialang.org/en/v1/manual/types/broadcast/))

A common scenario in computing is that
4.7. BROADCASTING

- 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

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

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

```julia
y\_vec = \sin.(x\_vec)
```

More generally, if $f$ is any Julia function, then $f\_\text{broadcast}$ 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.

```julia
function \text{chisq}(k)
    @assert k > 0 
z = \text{randn}(k)
    return sum(z \rightarrow z^2, z) # same as `sum(x^2 for x in z)`
end
```

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

```julia
\text{chisq}(3)
```

```julia
1.7875052353489465
```

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

```julia
\text{chisq}(-2)
```
Let’s try this out on an array of integers, adding the broadcast

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

```julia
3-element Array{Float64,1}:
1.317059322022995
4.088175390859048
9.740503148794987
```

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

```julia
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
```

```julia
5-element Array{Float64,1}:
2.1585290151921033
5.090702573174318
7.85887991940133
10.756802495307928
13.958924274663138
```

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

```julia
@ z = x + y - sin(x)
```

```julia
5-element Array{Float64,1}:
2.1585290151921033
5.090702573174318
7.85887991940133
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.

```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
```

```julia
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 Ref to fix them, or else it will try to broadcast the value.

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

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

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.

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.

4.8.1 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.
This would be roughly equivalent to

```
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
# run the function
```

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

```
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`
```

The scoping also applies to named arguments in functions.

```
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)
```

```
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
```

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

```
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)
```
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

```plaintext
\begin{verbatim}
[f(x) = x^2] # local `x` in scope
f(x) = a * x^2 # refers to the `a` in the outer scope
\end{verbatim}

\begin{verbatim}
5-element Array{Int64,1}:
  1
  4
  9
 16
 25
\end{verbatim}
```

### 4.8.2 Closures

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

```plaintext
\begin{verbatim}
f(x, a) = a * x^2
f(1, 0.2)
\end{verbatim}
```

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

```plaintext
\begin{verbatim}
a = 0.2
f(x) = a * x^2 # refers to the `a` in the outer scope
f(1) # univariate function
\end{verbatim}
```

```plaintext
\begin{verbatim}
0.2
\end{verbatim}
```

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

```plaintext
\begin{verbatim}
function g(a)
  f(x) = a * x^2 # refers to the `a` passed in the function
  f(1) # univariate function
end
g(0.2)
\end{verbatim}
```
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 \(f(x) = x^2, g(a) = 2a, h(a, b) = a + b\)

```julia
function solvemodel(x)
a = x^2
b = 2 * a
c = a + b
    return (a = a, b = b, c = c)  # note local scope of tuples!
end
solvemodel(0.1)
```

```
(a = 0.010000000000000002, b = 0.020000000000000004, c = 0.030000000000000006)
```

### 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
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 \texttt{Expectations.jl}, where we can pass a function to the \texttt{expectation} function.

```julia
using Expectations, Distributions
@show d = Exponential(2.0)
```

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

```plaintext
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

8.0
```

You can create and define using `function` as well

```plaintext
function snapabove(g, a)
    function f(x)
        if x > a  # "a" is captured in the closure f
            return g(x)
        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
g(fmt=:png);
plot(h, 0.0:0.1:3.0)
```

```
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

\[ \begin{align*}
    &\text{for } i \text{ in } 1:2 \quad \# \text{ introduces local } i \\
    &\quad \text{dval1 } = i \\
    &\quad \text{println}(i) \\
    &\text{end} \\
    &\quad \# \text{ @show (i, dval1) } \quad \# \text{ would fail as neither exists in this scope} \\
    &\text{for } i \text{ in } 1:2 \quad \# \text{ introduces a different local } i \\
    &\quad \text{println}(i) \\
    &\text{end}
\end{align*} \]

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

\[ \begin{align*}
    &\text{dval2 } = 0 \quad \# \text{ introduces variables} \\
    &\text{for } i \text{ in } 1:2 \quad \# \text{ introduces local } i \\
    &\quad \text{dval2 } = i \quad \# \text{ refers to outer variable} \\
    &\text{end} \\
    &\text{dval2} \quad \# \text{ still can't refer to `i`}
\end{align*} \]
Similarly, for while loops

```markdown
[103]:

```
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 example

```markdown
[104]:

```
x = 2.0
f(y) = x + y
z = f(4.0)
for i in 1:3
    z += i
end
println("z = "$z)
```

z = 12.0

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

```markdown
[105]:

```
function 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 \texttt{x\_vals} and \texttt{y\_vals} of equal length, compute their inner product using \texttt{zip()}.
Part 2: Using a comprehension, count the number of even numbers between 0 and 99.

- Hint: \texttt{iseven} returns \texttt{true} for even numbers and \texttt{false} for odds.

Part 3: Using a comprehension, take \texttt{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 Eq. (1) given a point \(x\) 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{uppercase("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 \( f \) mapping some interval \([a, b]\) into \( \mathbb{R} \).
- two scalars \( a \) and \( b \) providing the limits of this interval.
- An integer \( n \) determining the number of grid points.
- A number \( x \) satisfying \( a \leq x \leq b \).

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

Aim for clarity, not efficiency.
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:

\begin{verbatim}
[106]: open("us_cities.txt", "w") do f
    write(f, "new york: 8244910
los angeles: 3819702
chicago: 2787120
houston: 2145146
philadelphia: 1536471
phoenix: 1469471
san antonio: 1359758
san diego: 1326179
dallas: 1223229")
end
\end{verbatim}

Write a program to calculate total population across these cities.

Hints:

- If \( f \) is a file type then \texttt{eachline(f)} provides an iterable that steps you through the lines in the file.
- \texttt{parse(Int, "100")} converts the string \texttt{"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 \( \text{nodes} = [-1.0:0.5:1.0] \).

2. Instead of the \texttt{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 \texttt{function linapprox(f, a, b, n, x)}, it should be called as \texttt{function linapprox(f, nodes, x)}. * \texttt{step(nodes)}, \texttt{length(nodes)}, \texttt{nodes[1]}, and \texttt{nodes[end]} may be useful. * Type \texttt{?÷} 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

\[
\begin{align*}
\text{x_vals} &= [1, 2, 3] \\
\text{y_vals} &= [1, 1, 1] \\
\sum(x \cdot y \text{ for } (x, y) \text{ in zip(x_vals, y_vals)})
\end{align*}
\]

\[6\]

Part 2 solution:

One solution is

\[
\sum(\text{iseven, 0:99})
\]

\[50\]

Part 3 solution:

Here’s one possibility

\[
\begin{align*}
\text{pairs} &= ((2, 5), (4, 2), (9, 8), (12, 18)) \\
\sum(xy \rightarrow \text{all(iseven, xy)}, \text{pairs})
\end{align*}
\]

\[2\]

4.10.2 Exercise 2

\[
\text{p(x, coeff) = sum(a * x^((i-1)) \text{ for } (i, a) \text{ in enumerate(coeff))}
\]
4.10.3 Exercise 3

Here’s one solutions:

```python
function f_ex3(string)
    count = 0
    for letter in string
        if (letter == uppercase(letter)) && isletter(letter)
            count += 1
        end
    end
    return count
end
f_ex3("The Rain in Spain")
```

4.10.4 Exercise 4

Here’s one solutions:

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

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

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

```python
f_ex4_2(seq_a, seq_b) = Set(seq_a) ⊆ Set(seq_b)  # \subseteq (⊆) is unicode for ‘issubset’
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

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

Let's test it

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

```julia
x_grid = range(-1.0, 1.0, length = 100)
y_vals = f_ex5.(x_grid)
y = g_ex5.(x_grid)
plot(x_grid, y_vals, label = "true")
plot!(x_grid, y, label = "approximation")
```
4.10.6  

Exercise 6

```python
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

```plaintext
[1]: using InstantiateFromURL
github_project("QuantEcon/quantecon-notebooks-julia", version = "0.5.0")
# github_project("QuantEcon/quantecon-notebooks-julia", version = "0.5.0", instantiate = true) # uncomment to force package installation

[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

```plaintext
[3]: a = [10, 20, 30]

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

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

[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

```plaintext
[5]: typeof(randn(100))

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

```sh
[6]: ndims(a)
1
[7]: size(a)
(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

```sh
[8]: Array{Int64, 1} == Vector{Int64}
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

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

```sh
[10]: [1 2 3]  # a row vector is 2-dimensional
```

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

```plaintext
zeros(3)
```

This generalizes to matrices and higher dimensional arrays

```plaintext
zeros(2, 2)
```

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

```plaintext
fill(5.0, 2, 2)
```

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

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

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

```plaintext
fill(0, 2, 2)  # fills with 0, not 0.0
```

Or fill with a boolean type
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

```
x = [1, 2, 3]
y = x
y[1] = 2
x
```

```
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

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

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

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

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

```
3-element Array{Int64,1}:
140217924545424
140218264191104
140217924545456
```

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

```plaintext
2×2 Array{Int64,2}:
140218308386192  140218310206896
140217944844976   0
```

Manual Array Definitions

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

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

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

In two dimensions we can proceed as follows

```plaintext
a = [10 20 30 40]  # two dimensional, shape is 1 x n
```

```plaintext
1×4 Array{Int64,2}:
10 20 30 40
```

```plaintext
ndims(a)
```

```plaintext
2
```

```plaintext
a = [10; 20; 30; 40]  # 2 x 2
```

```plaintext
2×2 Array{Int64,2}:
10 20
30 40
```

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

```plaintext
a = [10; 20; 30; 40]
```
5.3. ARRAY BASICS

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

[27]: ndims(a)

[27]: 1

Instead transpose the matrix (or adjoint if complex)

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

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

[29]: ndims(a)

[29]: 2

5.3.3 Array Indexing

We’ve already seen the basics of array indexing

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

[30]: 30

[31]: a[1:3]

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

For 2D arrays the index syntax is straightforward

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

[32]: -1.1732480351221797

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

```julia
a = randn(2, 2)
b = [true false; false true]
```

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

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

```julia
a = zeros(4)
a[2:end] .= 42
```
5.3. ARRA Y BASICS

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.

```
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.

```
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.

```

```
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.

```

```
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.

```

```
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.

```

```
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.

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

Similarly

\[
a = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}
b = a'  \quad \# \text{ transpose}
\]

\[
\text{typeof(b)}
\]

\[
\text{Adjoint}\{\text{Int64,Array}\{\text{Int64,2}\}}
\]

To copy into a dense array

\[
a = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}
b = a'  \quad \# \text{ transpose}
c = \text{Matrix}(b)  \quad \# \text{ convert to matrix}
d = \text{collect}(b)  \quad \# \text{ also collect’ works on any iterable}
c == d
\]

\[
\text{true}
\]

### 5.3.5 Special Matrices

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

As an example, consider creating a diagonal matrix

\[
d = [1.0, 2.0]
a = \text{Diagonal}(d)
\]

\[
\text{2}\times\text{2 Diagonal}\{\text{Float64,Array}\{\text{Float64,1}\}}:
\begin{bmatrix}
1.0 & 0 \\
0 & 2.0
\end{bmatrix}
\]

As you can see, the type is \texttt{2\times2 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 \texttt{Diagonal} are just as much a “matrix” as the dense matrices we have using (see the introduction to types lecture for more)
2a = [2.0 0.0; 0.0 4.0]
b * a = [0.7150607666265514 0.16189616391105188; 0.8880266570758839 1.1683586165440394]

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

\[
\begin{bmatrix}
1.0 & 2.0 \\
3.0 & 3.0
\end{bmatrix}
\]

# poor style, inefficient code

Whereas you should instead use

\[
\begin{bmatrix}
1.0 & 2.0 \\
3.0 & 3.0
\end{bmatrix}
\]

# good style, and note the lack of dimensions of I

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

```
typeof(I)
```

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.

```
x = [1 2 3]
y = x # name ‘y’ binds to whatever value ‘x’ bound to
```

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

```
x = [1 2 3] # name ‘y’ binds to whatever ‘x’ bound to
y = x
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.

```julia
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

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

```julia
2-element Array{Int64,1}:
5  11
```

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

```julia
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
```

```julia
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
5.3. ARRA Y BASICS

```plaintext
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
```

2-element Array{Int64,1}:
4294967297
140218149294272

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

```plaintext
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
```

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

```plaintext
x = 2

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

```plaintext
x = 2
```

This is also true for other immutable types such as tuples, as well as some vector types

```plaintext
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!"
```
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

```julia
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)

To sort an array

```julia
b = sort(a, rev = true)  # returns new array, original not modified
```

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

```julia
b = sort!(a, rev = true)  # returns *modified original* array
```

3-element Array{Int64,1}:
1
0
-1
5.4. OPERATIONS ON ARRAYS

5.4.2 Matrix Algebra

For two dimensional arrays, * means matrix multiplication

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

\[ 1\times2 \text{ Array}\{\text{Float64},2\}: \]
\[ 1.0 \quad 1.0 \]

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

\[ 2\times2 \text{ Array}\{\text{Float64},2\}: \]
\[ 1.0 \quad 1.0 \]
\[ 1.0 \quad 1.0 \]

\[ a \ast b \]

\[ 1\times2 \text{ Array}\{\text{Float64},2\}: \]
\[ 2.0 \quad 2.0 \]

\[ b \ast a' \]

\[ 2\times1 \text{ Array}\{\text{Float64},2\}: \]
\[ 2.0 \]
\[ 2.0 \]

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

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

\[ 2\times2 \text{ Array}\{\text{Int64},2\}: \]
\[ 1 \quad 2 \]
\[ 2 \quad 3 \]

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

\[ 2\times2 \text{ Array}\{\text{Float64},2\}: \]
\[ 1.0 \quad 1.0 \]
\[ 1.0 \quad 1.0 \]
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.
If you want an inner product in this setting use `dot()` or the unicode `\cdot<TAB>`

```
[75]:
dot(ones(2), ones(2))
```

```
[75]:
2.0
```

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

\[
\begin{align*}
\text{ones}(2, 2) \cdot \text{ones}(2, 2) \quad &\text{# matrix multiplication} \\
\end{align*}
\]

\[
\begin{align*}
\text{ones}(2, 2) \times \text{ones}(2, 2) \\
\end{align*}
\]
5.4. OPERATIONS ON ARRAYS

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

[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

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

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

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

[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.

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

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

Scalar multiplication is similar

[84]:
A = ones(2, 2)

[84]:
2×2 Array{Float64,2}:
1.0  1.0
1.0  1.0

[85]:
2 * A  # same as 2 .* A

[85]:
2×2 Array{Float64,2}:
2.0  2.0
2.0  2.0

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

Unlike MATLAB and other languages, scalar addition requires the .+ in order to correctly broadcast

[86]:
x = [1, 2]
x .+ 1  # not x + 1
x .- 1  # not x - 1
```
Elementwise Comparisons

Elementwise comparisons also use the .x style notation

```plaintext
[87]: a = [10, 20, 30]
```

```plaintext
[87]: 3-element Array{Int64,1}:
    10
    20
    30
```

```plaintext
[88]: b = [-100, 0, 100]
```

```plaintext
[88]: 3-element Array{Int64,1}:
    -100
     0
     100
```

```plaintext
[89]: b .> a
```

```plaintext
[89]: 3-element BitArray{1}:
    0
    0
    1
```

```plaintext
[90]: a .== b
```

```plaintext
[90]: 3-element BitArray{1}:
    0
    0
    0
```

We can also do comparisons against scalars with parallel syntax

```plaintext
[91]: b
```

```plaintext
[91]: 3-element Array{Int64,1}:
    -100
     0
     100
```

```plaintext
[92]: 1
```

```plaintext
[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
5.4. OPERATIONS ON ARRAYS

```plaintext
[93]: a = randn(4)

4-element Array{Float64,1}:
0.5130793847452865
0.6359455761384935
-0.15452390070466798
0.1918902530079361

[94]: a .< 0

4-element BitArray{1}:
0
0
1
0

[95]: a[a .< 0]

1-element Array{Float64,1}:
-0.15452390070466798
```

Changing Dimensions

The primary function for changing the dimensions of an array is **reshape()**

```plaintext
[96]: a = [10, 20, 30, 40]

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

[97]: b = reshape(a, 2, 2)

2×2 Array{Int64,2}:
10 30
20 40
```

Notice that this function returns a view on the existing array.

This means that changing the data in the new array will modify the data in the old one.

```plaintext
[98]: b

2×2 Array{Int64,2}:
10 30
20 40

[99]: b[1, 1] = 100  # continuing the previous example

100
```
To collapse an array along one dimension you can use `dropdims()`

```
[102]: a = [1 2 3 4]  # two dimensional
[102]: 1×4 Array{Int64,2}:
 1 2 3 4
[103]: dropdims(a, dims = 1)
[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 `log`, `exp`, `sin`, etc.

```
[104]: log(1.0)
[104]: 0.0
```

By default, these functions act *elementwise* on arrays

```
[105]: log.(1:4)
[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

```
[106]: [ log(x) for x in 1:4 ]
```
5.5. RANGES

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

```
A = [1 2; 3 4]
```

```
2×2 Array{Int64,2}:
1 2
3 4
```

```
det(A)
```

```
-2.0
```

```
tr(A)
```

```
5
```

```
eigvals(A)
```

```
2-element Array{Float64,1}:
-0.3722813232690143
5.372281323269014
```

```
r(A)
```

```
2
```

5.5 Ranges

As with many other types, a Range can act as a vector.

```
a = 10:12  # a range, equivalent to 10:1:12
@show Vector(a)  # can convert, but shouldn't
b = Diagonal([1.0, 2.0, 3.0])
b .- [1.0; 2.0; 3.0]
```

Vector(a) = [10, 11, 12]
3-element Array{Float64,1}:
  9.0
  20.0
  33.0

Ranges can also be created with floating point numbers using the same notation.

```
0.0:0.1:1.0  # 0.0, 0.1, 0.2, ... 1.0
```

But care should be taken if the terminal node is not a multiple of the set sizes.

```
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., `linspace` in other languages

```
maxval = 1.0
minval = 0.0
numpoints = 10
a = range(minval, maxval, length=numpoints)
# or range(minval, stop=maxval, length=numpoints)
maximum(a) == maxval
```

### 5.6 Tuples and Named Tuples

(See tuples and named tuples documentation)

We were introduced to tuples earlier, which provide high-performance immutable sets of distinct types.

```
t = (1.0, "test")  # access by index
t[1] = 3.0  # would fail as tuples are immutable
a, b = t  # unpack
println("a = ", a, " and b = ", b)  # access by index

a = 1.0 and b = test
```

As well as named tuples, which extend tuples with names for each argument.

```
t = (val1 = 1.0, val2 = "test")
t.val1  # access by index
# a, b = t  # bad style, better to unpack by name with @unpack
```
5.6. TUPLES AND NAMED TUPLES

```
println("val1 = $(t.val1) and val1 = $(t.val1)") # access by name
val1 = 1.0 and val1 = 1.0
```

While immutable, it is possible to manipulate tuples and generate new ones

```
t2 = (val3 = 4, val4 = "test!!")
t3 = merge(t, t2) # new tuple
```

```
(vall = 1.0, val2 = "test", val3 = 4, val4 = "test!!")
```

Named tuples are a convenient and high-performance way to manage and unpack sets of parameters

```
function f(parameters)
  α, β = parameters.α, parameters.β # poor style, error prone if adding parameters
  return α + β
end
parameters = (α = 0.1, β = 0.2)
f(parameters)
```

```
0.30000000000000004
```

This functionality is aided by the `Parameters.jl` package and the `@unpack` macro

```
using Parameters

function f(parameters)
  @unpack α, β = parameters # good style, less sensitive to errors
  return α + β
end
parameters = (α = 0.1, β = 0.2)
f(parameters)
```

```
0.30000000000000004
```

In order to manage default values, use the `@with_kw` macro

```
using Parameters
paramgen = @with_kw (α = 0.1, β = 0.2) # create named tuples with defaults

# creates named tuples, replacing defaults
@show paramgen() # calling without arguments gives all defaults
@show paramgen(α = 0.2)
@show paramgen(α = 0.2, β = 0.5);
```

```
paramgen() = (α = 0.1, β = 0.2)
paramgen(α=0.2) = (α = 0.2, β = 0.2)
paramgen(α=0.2, β=0.5) = (α = 0.2, β = 0.5)
```

An alternative approach, defining a new type using `struct` tends to be more prone to accidental misuse, and leads to a great deal of boilerplate code.
CHAPTER 5. ARRAYS, TUPLES, RANGES, AND OTHER FUNDAMENTAL TYPES

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

```
Type: 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

```
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 = \$x")
    end
    x
end

@show f(1.0)
@show f(-1.0);
```

```
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.
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) failed")
else
    println("f(x) successful");
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

function f(x)
    x > 0.0 ? sqrt(x) : nothing # the "a ? b : c" pattern is the ternary end

f(1.0)

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 \texttt{Union\{Nothing, Float64\}} – that is, it returns either a floating point or a \texttt{nothing}.

You will see this type directly if you use an array containing both types

x = [1.0, nothing]

2-element Array{Union{Nothing, Float64},1}:
1.0 nothing

When considering error handling, whether you want a function to return \texttt{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

x = [0.1, -1.0, 2.0, -2.0]
y = f.(x)
# presumably check `y`
4-element Array{Union{Nothing, Float64},1}:
0.31622776601683794
nothing
1.4142135623730951
nothing

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.

```plaintext
function f(x)
    @assert x > 0.0
    sqrt(x)
end
f(1.0)
```

1.0

Finally, **nothing** is a good way to indicate an optional parameter in a function

```plaintext
function f(x; z = nothing)
    if isnothing(z)
        println("No z given with $x")
    else
        println("z = $z given with $x")
    end
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.

```plaintext
function f(x)
    if x > 0.0
        return x
    else
        return NaN
    end
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

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.

```plaintext
# 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

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.

```plaintext
# 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.

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.

```plaintext
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)

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

```plaintext
x = [3.0, missing, 5.0, missing, missing]
```

A key feature of `missing` is that it propagates through other function calls - unlike `nothing`

```plaintext
f(x) = x^2
@show missing + 1.0
@show missing * 2
@show missing * "test"
@show f(missing);  # even user-defined functions
@show mean(x);
```

```plaintext
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

```plaintext
x = missing
@show x == missing
@show x === missing  # an exception
@show ismissing(x);
```

```plaintext
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`.

```plaintext
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 Eq. (1) that \( \{S_t\} \) obeys

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

(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 Eq. (2) given \( A \) and \( \Sigma \).

To test your solution, observe that the limit \( S \) is a solution to the matrix equation

\[ S = ASA' + Q \]  where \( Q := \Sigma \Sigma' \)  

(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 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 \( 1 \leq \tau \leq T \)

\[
\frac{1}{\tau} \sum_{t=1}^{\tau} y_T
\]

- 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 \( y^n_T \).
- 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.

\[
\sum_{n=1}^{N} \frac{y^n_T}{N} \quad \text{and} \quad \sum_{n=1}^{N} \left( \frac{y^n_T}{N} \right)^2 - \left( \sum_{n=1}^{N} \frac{y^n_T}{N} \right)^2.
\]

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 $a = 0.1, b = 0.2c = 0.5, d = 1.0, \sigma = 0.1$. * 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 handcodded iteration used in Exercise 1.

Hint: Convert the matrix to a vector to use `fixedpoint`. e.g. $A = \begin{bmatrix} 1 & 2; 3 & 4 \end{bmatrix}$ 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

```matlab
function compute_asymptotic_var(A, Σ; S0 = Σ * Σ', tolerance = 1e-6, maxiter = 500)
    V = Σ * Σ'
    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
```

Here’s the code:

```
A = \begin{bmatrix} 0.8 & -0.2; -0.1 & 0.7 \end{bmatrix}
Σ = \begin{bmatrix} 0.5 & 0.4; 0.4 & 0.8 \end{bmatrix}
```

Note that all eigenvalues of $A$ lie inside the unit disc.

```octave
maximum(abs, eigvals(A))
```

0.9

Let’s compute the asymptotic variance

```octave
our_solution = compute_asymptotic_var(A, Σ)
```

```octave
times(2, 2, Float64) = 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.

```octave
using QuantEcon

norm(our_solution - solve_discrete_lyapunov(A, Σ * Σ'))
```

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

[1]: using InstantiateFromURL
github_project("QuantEcon/quantecon-notebooks-julia", version = "0.5.0")
# github_project("QuantEcon/quantecon-notebooks-julia", version = "0.5.0", instantiate =false) # uncomment to force package installation
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.

```julia
@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.

```julia
x = 1
typeof(x)
```

```
 typeof(x) = Int64
```

The name `x` binds to the value `1`, created as a literal.

6.3.2 Parametric Types

(See [parametric types documentation](https://docs.julialang.org/en/v1/stdlib/)).

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

```julia
@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`:

```plaintext
x = (1, 2.0, "test")
@show typeof(x)
```

\[
\text{typeof}(x) = \text{Tuple}\{\text{Int64}, \text{Float64}, \text{String}\}
\]

Another tuple example:

```plaintext
x = (a = 1, b = 2.0, c = "test")
@show typeof(x)
```

\[
\text{typeof}(x) = \text{NamedTuple}\{(:a, :b, :c), \text{Tuple}\{\text{Int64}, \text{Float64}, \text{String}\}\}
\]

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.

```plaintext
typeof(:a)
```

\[
\text{Symbol}
\]

**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.

\[ x = 42 \]
\[ \@show \text{typeof}(x); \]

\text{typeof}(x) = \text{Int64}

Thus, \( x \) is just a symbol bound to a value of type \text{Int64}.

We can \textit{rebind} the symbol \( x \) to any other value, of the same type or otherwise.

\[ x = 42.0 \]

\[ 42.0 \]

Now \( x \) “points to” another value, of type \text{Float64}

\[ \text{typeof}(x) \]

\[ \text{Float64} \]

However, beyond a few notable exceptions (e.g. \textit{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 \textit{abstract types documentation})

Up to this point, most of the types we have worked with (e.g., \texttt{Float64}, \texttt{Int64}) are examples of \textit{concrete types}.

Concrete types are types that we can \textit{instantiate} – i.e., pair with data in memory.

We will now examine \textit{abstract types} that cannot be instantiated (e.g., \texttt{Real}, \texttt{AbstractFloat}).

For example, while you will never have a \texttt{Real} number directly in memory, the abstract types help us organize and work with related concrete types.
### 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.

```julia
@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`

```julia
@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.

```julia
Number <: Any
```

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.

```julia
using Base: show_supertypes # import the function from the `Base` package
show_supertypes(Int64)
```

Int64 <: Signed <: Integer <: Real <: Number <: Any
And the `subtypes` which gives a list of the available subtypes for any packages or code currently loaded:

```julia
@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:

```julia
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}
```

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.

```julia
f(y) = 2y # define some function
x = [1, 2, 3]
z = f(x) # call with an integer array - compiler deduces type
```

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

### 6.5.1 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.
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.

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:

```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])

2-element Array{Float64,1}:
  9.1
 14.3

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])

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

@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]

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

```plaintext
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

```plaintext
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

```plaintext
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
```

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

\[
\begin{aligned}
\text{struct } \text{Foo2} & \quad \text{a} :: \text{Float64} \\
& \quad \text{b} :: \text{Integer} \quad \# \text{BAD! Not a concrete type} \\
& \quad \text{c} :: \text{Vector} \{ \text{Real} \} \quad \# \text{BAD! Not a concrete type} \\
\end{aligned}
\]

The second issue is that by choosing a type (as in the `Foo` above), you may be unnecessarily constraining what is allowed

\[
\begin{aligned}
f(x) &= x.a + x.b + \text{sum}(x.c) \quad \# \text{use the type} \\
a &= 2.0 \\
b &= 3 \\
c &= [1.0, 2.0, 3.0] \\
\text{foo} &= \text{Foo}(a, b, c) \\
\text{@show } f(\text{foo}) \quad \# \text{call with the foo, no problem} \\
\text{# some other typed for the values} \\
a &= 2 \quad \# \text{not a floating point but } f() \text{ would work} \\
b &= 3 \\
c &= [1.0, 2.0, 3.0] \quad \# \text{transpose is not a } \text{Vector} \text{ but } f() \text{ would work} \\
\text{# foo} &= \text{Foo}(a, b, c) \quad \# \text{fails to compile} \\
\text{# works with } \text{`NotTyped`} \text{ version, but low performance} \\
\text{foo}_\text{nt} &= \text{FooNotTyped}(a, b, c) \\
\text{@show } f(\text{foo}_\text{nt});
\end{aligned}
\]

\[
\begin{aligned}
f(\text{foo}) &= 11.0 \\
f(\text{foo}_\text{nt}) &= 11.0
\end{aligned}
\]

### 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.

\[
\begin{aligned}
\text{struct } \text{Foo3} & \{ T1, T2, T3 \} \\
& \quad \text{a} :: T1 \quad \# \text{could be any type} \\
& \quad \text{b} :: T2 \\
& \quad \text{c} :: T3 \\
\end{aligned}
\]

\[
\begin{aligned}
\text{# works fine} \\
a &= 2 \\
b &= 3 \\
c &= [1.0, 2.0, 3.0] \quad \# \text{transpose is not a } \text{Vector} \text{ but } f() \text{ would work} \\
\text{foo} &= \text{Foo3}(a, b, c) \\
\text{@show } \text{typeof}(\text{foo}) \\
f(\text{foo})
\end{aligned}
\]

\[
\text{typeof}(\text{foo}) = \text{Foo3}\{\text{Int64},\text{Int64},\text{Adjoint}\{\text{Float64},\text{Array}\{\text{Float64},1\}\}\}
\]
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

```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)
```

typeof(foo) = Foo4{Int64,Int64,Adjoint{Float64,Array{Float64,1}}}

This ensures that

- \( a \) and \( b \) are a subtype of \texttt{Real}, and \( + \) in the definition of \( f \) works
- \( c \) is a one dimensional abstract array of \texttt{Real} values

The code works, and is equivalent in performance to a \texttt{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 \texttt{Parameters.jl} library.

```julia
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 values

@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]

[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 autodifferentiation, 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.

```julia
# BAD
x = [5.0, 6.0, 2.1]

function g(x::Array{Float64, 1}) # not generic!
y = zeros(length(x)) # not generic, hidden float!
z = Diagonal(ones(length(x))) # 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)
```

[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

```
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)])
```

2-element Array{BigInt,1}:

1
4

- Use `typeof` or `eltype` to declare a type

```
@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

```
@show typeof(ones(3))
@show typeof(ones(Int64, 3))
@show typeof(zeros(Int64, 3));

typeof(ones(3)) = Array{Float64,1}
typeof(ones(Int64, 3)) = Array[Int64,1]
typeof(zeros(3)) = Array{Float64,1}
typeof(zeros(Int64, 3)) = Array[Int64,1]
```

- Use `one` and `zero` to write generic code

```
@show typeof(1)
@show typeof(1.0)
@show typeof(BigFloat(1.0))
@show typeof(one(BigFloat))    # gets multiplicative identity, passing in type
@show typeof(zero(BigFloat))

x = BigFloat(2)

@show typeof(one(x))    # can call with a variable for convenience
@show typeof(zero(x));

typeof(1) = Int64
typeof(1.0) = Float64
This last example is a subtle, because of something called type promotion

- Assume reasonable type promotion exists for numeric types

```julia
[37]: # ACCEPTABLE
    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

```julia
[38]: # BAD
    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
            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)  # any type with an additive identity
        if x > zero(x)
            return x + one(x)  # more general but less important of a change
        otherwise
            return zero(x)
        end
    end

    @show typeof(g3(x))
    @show typeof(g3(x2));  # type stable
```

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

```plaintext
@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

```plaintext
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
```

```plaintext
@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

```plaintext
x = -2//3  # `Rational` number, -2/3
@show typeof(x)
@show ourabs(x);
```

```
```
6.7. INTRODUCTION TO MULTIPLE DISPATCH

\[
\text{typeof}(x) = \text{Rational}\{\text{Int64}\}
\]
\[
\text{ourabs}(x) = 2/3
\]

You will also note that we used an abstract type, \texttt{Real}, and an incomplete parametric type, \texttt{Complex}, when defining the above functions.

Unlike the creation of \texttt{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 \texttt{AbstractRange}, which we can analyze with \texttt{typeof}, \texttt{supertype} and \texttt{show_supertypes}.

```julia
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))
```

\[
\text{typeof}(x) = \text{StepRangeLen}\{\text{Float64}, \text{Base.TwicePrecision}\{\text{Float64}\}, \text{Base.TwicePrecision}\{\text{Float64}\}}
\]
\[
\text{typeof}(x_2) = \text{StepRange}\{\text{Int64}, \text{Int64}\}
\]
\[
\text{supertype}(\text{typeof}(x)) = \text{AbstractRange}\{\text{Float64}\}
\]

To see the entire tree about a particular type, use \texttt{show_supertypes}.

```julia
show_supertypes(typeof(x))  # or typeof(x) |> show_supertypes
```

\[
\text{StepRangeLen}\{\text{Float64}, \text{Base.TwicePrecision}\{\text{Float64}\}, \text{Base.TwicePrecision}\{\text{Float64}\}} \begin{array}{c}
\leq \text{AbstractRange}\{\text{Float64}\}
\end{array} \begin{array}{c}
\leq \text{AbstractArray}\{\text{Float64}, 1\}
\end{array} \begin{array}{c}
\leq \text{Any}
\end{array}
\]
The types of the range objects can be very complicated, but are both subtypes of `AbstractRange`.

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`.

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.

```
[49]: slopes(f_x::AbstractVector, x::AbstractRange) = diff(f_x) / step(x)

[49]: slopes (generic function with 1 method)

We can use auto-differentiation to compare the results.

```
using Plots, ForwardDiff

# 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")
```

[50]:
Consider a variation where we pass a function instead of an `AbstractArray`

```
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];
```

```
typeof(q) <: Function = true
typeof(x) <: AbstractRange = true
q_slopes_x[1] = 0.9983341664682815
```

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

```
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 $N = 15$.

```julia
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)
```

78.215 ns (1 allocation: 112 bytes)
659.635 ns (5 allocations: 1.98 KiB)

```julia
3×3 Array{Float64,2}:
-1.96261 -0.0944077 2.34979
-0.438536 2.25694 -0.933619
3.08412 -2.09073 0.0314215
```

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

```julia
Σ = [0.4 0.3;
    0.3 0.45]
G = I
R = 0.5 * Σ
gain(Σ, G, R) = Σ * G' * inv(G * Σ * G' + R)
@btime gain($Σ, $G, $R)
```

795.419 ns (10 allocations: 1.94 KiB)

```julia
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.

```julia
using Polynomials
p = Poly([2, -5, 2], :x)  # :x just gives a symbol for display
@show p
p′ = polyder(p)  # gives the derivative of p, another polynomial
@show p(0.1), p′(0.1)  # call like a function
@show roots(p);  # find roots such that p(x) = 0
```

\[
p = \text{Poly}(2 - 5x + 2x^2)
\]
\[
(p(0.1), p′(0.1)) = (1.52, -4.6)
\]
\[
\text{roots}(p) = [0.5, 2.0]
\]

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 polyder function.

The signature of the function should be `newtonsmethod(p::Poly, x_0; tolerance = 1E-7, maxiter = 100)`, where `p::Poly` 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 `roots(p)` function.

6.8.5 Exercise 5 (Advanced)

The trapezoidal rule approximates an integral with

\[
\int_a^b f(x) \, dx \approx \sum_{n=1}^{N} \frac{f(x_{n-1}) + f(x_n)}{2} \Delta x_n
\]

where \(x_0 = a\), \(x_N = b\), 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

- `trapezoidal(f, x)` for any `typeof(x) = AbstractArray` and `typeof(f) == AbstractArray` where `length(x) = length(f)`
- `trapezoidal(f, x)` for any `typeof(x) = AbstractRange` and `typeof(f) == AbstractArray` where `length(x) = length(f)`
  
  - Exploit the fact that `AbstractRange` has constant step sizes to specialize the algorithm
- `trapezoidal(f, x, k, N)` where `typeof(f) = Function`, and the other arguments are `Real`
– For this, build a uniform grid with \( N \) points on \([\bar{x}, \bar{x}]\) – call the \( f \) function at those grid points and use the existing trapezoidal\((f, x)\) from the implementation

With these: 1. Test each variation of the function with \( f(x) = x^2 \) with \( \underline{x} = 0, \bar{x} = 1 \). 2. From the analytical solution of the function, plot the error of \text{trapezoidal}(f, \underline{x}, \bar{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

\[\text{using QuadGK}\]
\[f(x) = x^2\]
\[\text{value, accuracy} = \text{quadgk}(f, 0.0, 1.0)\]
\[\text{(0.3333333333333333, 5.551115123125783e-17)}\]

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

\[\frac{d}{d\bar{x}} \int_{\underline{x}}^{\bar{x}} f(x) dx\]

Hint: See the following code for the general pattern, and be careful to follow the rules for generic programming.

\[\text{using ForwardDiff}\]
\[\text{function f(a, b; N = 50)}\]
\[\text{r = range(a, b, length=N) # one}\]
\[\text{return mean(r)}\]
\[\text{end}\]
\[Df(x) = \text{ForwardDiff}.\text{derivative}(y \rightarrow f(0.0, y, x))\]
\[@\text{show} f(0.0, 3.0)\]
\[@\text{show} f(0.0, 3.1)\]
\[Df(3.0)\]

\[f(0.0, 3.0) = 1.5\]
\[f(0.0, 3.1) = 1.55\]

\[0.5\]
CHAPTER 6. INTRODUCTION TO TYPES AND GENERIC PROGRAMMING
Part II

Packages and Software Engineering in Julia
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*?

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

```julia
[1]:
using InstantiateFromURL
github_project("QuantEcon/quantecon-notebooks-julia", version = "0.5.0")
# github_project("QuantEcon/quantecon-notebooks-julia", version = "0.5.0", instantiate = false) # uncomment to force package installation

[2]:
using LinearAlgebra, Statistics
using Distributions, Plots, QuadGK, Polynomials, Interpolations
```

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.

```julia
[3]:
using Distributions
x = 1
y = Normal()
z = "foo"
@show x, y, z
@show typeof(x), typeof(y), typeof(z)
@show supertype(typeof(x))
```
7.3. EXPLORING TYPE TREES

# pipe operator, |>, is is equivalent
@show typeof(x) |> supertype
@show supertype(typeof(y))
@show typeof(z) |> supertype
@show typeof(x) <: Any;

(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

[4]: using Base: show_supertypes # import the function from the `Base` package

show_supertypes(Int64)

Int64 <: Signed <: Integer <: Real <: Number <: Any

[5]: subtypes(Integer)

[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`

[6]: # from https://github.com/JuliaLang/julia/issues/24741

```julia
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
```

[6]: subtypetree (generic function with 3 methods)

Applying this to `Number`, we see the tree of types currently loaded

[7]: subtypetree(Number) # warning: do not use this function on `Any`!

Number
    Complex
    Real
    AbstractFloat
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`

```
# 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
```

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

```
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

```
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

```
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`

```
@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;
```

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.
- 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 abstact 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

```julia
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

```julia
@show rand(d1);
```

```julia
rand(d1) = 1.542299859157156
```
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.

```julia
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, -0.8970953926873437, 4.104736084254901, 3.171988193214245, 2.672207523507183, 5.7623809141775935]

The \texttt{Sampleable\{Univariate,Continuous\}} and, especially, the \texttt{Sampleable\{Multivariate,Continuous\}} abstract types are useful generic interfaces for Monte-Carlo and Bayesian methods.

Moving down the tree, the \texttt{Distributions\{Univariate, Continuous\}} abstract type has other functions we can use for generic algorithms operating on distributions.

These match the mathematics, such as \texttt{pdf}, \texttt{cdf}, \texttt{quantile}, \texttt{support}, \texttt{minimum}, \texttt{maximum}, etc.

```julia
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) = 0.632120588285577
support(d1) = RealInterval(-Inf, Inf)
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

```julia
using StatsPlots

d = Normal(2.0, 1.0)
plot(d) # note no other arguments!
```

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

```julia
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

```
[20]: d = OurTruncatedExponential(1.0, 2.0)
   @show minimum(d), maximum(d)
   @show support(d) # why does this work?
```

\[(\text{minimum}(d), \text{maximum}(d)) = (0, 2.0)\]

\[\text{support}(d) = \text{RealInterval}(0.0, 2.0)\]

```
[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

\[\text{Distributions.support}(d::\text{Distribution}) = \text{RealInterval(}\text{minimum}(d), \text{maximum}(d))\]

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

```
[21]: plot(d) # uses the generic code!
```

```
[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.

\[ d = \text{Truncated}(\text{Exponential}(0.1), 0.0, 2.0) \]
\[ @show \text{typeof}(d) \]
\[ \text{plot}(d) \]

\text{typeof}(d) = \text{Truncated}\{\text{Exponential}\{\text{Float64}\}, \text{Continuous}\}

\[ d = \text{Truncated}(\text{Exponential}(0.1), 0.0, 2.0) \]
\[ @show \text{typeof}(d) \]
\[ \text{plot}(d) \]

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 ·, 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 pro-
vides motivation for the abstract **Number** type in Julia

• **Remark:** We use the term “motivation” because they are not formally con-
nected and the mapping is imperfect.
• The main difficulty when dealing with numbers that can be concretely cre-
ated 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 `typeof(a) = typeof(b) = T <: Number`, then under an informal definition of the
**generic interface** for **Number**, the following must be defined

• the additive operator: \( a + b \)
• the multiplicative operator: \( a \cdot b \)
• 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 **Number**, we can design algorithms using any of these functions and not concern ourselves with the par-
ticular concrete type.

Furthermore, that generality in designing algorithms comes with no compromises on perfor-
mance compared to carefully designed algorithms written for that particular type.

To demonstrate this for a complex number, where **Complex{Float64} <: Number**

```plaintext
a = 1.0 + 1.0im
b = 0.0 + 2.0im
@show typeof(a)
@show typeof(a) <: Number
@show a + b
@show -a
@show a - b
@show zero(a)
@show one(a);
```

typeof(a) = Complex{Float64}
typeof(a) <: Number = true
a + b = 1.0 + 3.0im
7.5. NUMBERS AND ALGEBRAIC STRUCTURES

\[ a \times b = -2.0 + 2.0\text{im} \]
\[-a = -1.0 - 1.0\text{im} \]
\[ a - b = 1.0 - 1.0\text{im} \]
\[ \text{zero}(a) = 0.0 + 0.0\text{im} \]
\[ \text{one}(a) = 1.0 + 0.0\text{im} \]

And for an arbitrary precision integer where \texttt{BigInt} <: \texttt{Number} (i.e., a different type than the \texttt{Int64} you have worked with, but nevertheless a \texttt{Number})

```plaintext
[24]:
  a = BigInt(10)
  b = BigInt(4)
  @show typeof(a)
  @show typeof(a) <: Number
  @show a + b
  @show a * b
  @show -a
  @show a - b
  @show zero(a)
  @show one(a);
```

```plaintext
typeof(a) = BigInt
typeof(a) <: Number = true
a + b = 14
a * b = 40
-a = -10
a - b = 6
zero(a) = 0
one(a) = 1
```

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 \texttt{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 <: \texttt{Number}, and is parameterized (e.g. \texttt{Complex\{Float64\}} is a complex number storing the imaginary and real parts in \texttt{Float64})

```plaintext
[25]:
x = 4.0 + 1.0\text{im}
@show x, typeof(x)
xbig = BigFloat(4.0) + 1.0\text{im}
@show xbig, typeof(xbig);
```

```plaintext
(x, typeof(x)) = (4.0 + 1.0im, Complex\{Float64\})
(xbig, typeof(xbig)) = (4.0 + 1.0im, Complex\{BigFloat\})
```

The implementation of the \texttt{Complex} numbers use the underlying operations of storage type, so as long as +, * etc. are defined – as they should be for any \texttt{Number} – the complex operation can be defined

```plaintext
[26]: @which +(x,x)
```

+\( (z::\text{Complex}, w::\text{Complex}) \text{ in Base at complex.jl:271} \)

Following that link, the implementation of + for complex numbers is

\(+ (z::\text{Complex}, w::\text{Complex}) = \text{Complex}(\text{real}(z) + \text{real}(w), \text{imag}(z) + \text{imag}(w)) \)

\( \text{real}(z) \) and \( \text{imag}(z) \) returns the associated components of the complex number in the underlying storage type (e.g. \text{Float64} or \text{BigFloat}).

The rest of the function has been carefully written to use functions defined for any \text{Number} (e.g. + but not <, since it is not part of the generic number interface).

To follow another example, look at the implementation of \text{abs} specialized for complex numbers

\[ \text{abs}(z::\text{Complex}) \text{ in Base at complex.jl:260} \]

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) \text{ where T<: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

\[ \text{abs}(x::\text{Float64}) = \text{abs}_\text{float}(x) \]

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}_\text{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: \( \text{inv}(a) \)
- the multiplicative inverse operation: \( a / b = a \cdot \text{inv}(b) \)
- 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 \( \frac{a}{b} \)

```plaintext
[30]:
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;
```

```plaintext
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

```plaintext
[31]: @show Float64 <: AbstractFloat
    @show BigFloat <: AbstractFloat
    @show eps(Float64)
    @show eps(BigFloat);
```

```plaintext
Float64 <: AbstractFloat = true
BigFloat <: AbstractFloat = true
eps(Float64) = 2.220446049250313e-16
eps(BigFloat) = 1.727233711018888925077270372560079914223200072887256277004740694033718360632485e-77
```

The `isless` function also has multiple methods.

First let’s try with integers

```plaintext
[32]: @which isless(1, 2)
```

```plaintext
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.
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

```plaintext
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
```

\[ \text{quadgk}(f, 0.0, 1.0) = (0.3333333333333333, 5.551115123125783e-17) \]
Of course, univariate polynomials are another type of univariate function

```
[35]: using Polynomials
    p = Poly([2, -5, 2], :x)  # :x just gives a symbol for display
   @show p
   @show p(1.0)  # call like a function
    plotfunctions(p)  # same generic function
```

```
p = Poly(2 - 5*x + 2*x^2)
p(1.0) = -1.0
```
Similarly, the result of interpolating data is also a function

```
[36]: using Interpolations
    x = 0.0:0.2:1.0
    f(x) = x^2
    f_int = 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 `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 writing generic code (e.g. continuous-time solutions for ODEs and matrix-free methods), but cannot be implemented rigorously since the `Matrix` type is a semi-group as well as an `AbstractArray`, but not all semi-groups are `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

$$\int \bar{x} f(x) \, dx \approx \omega \cdot \bar{f}$$

where \(\bar{f} \equiv [f(x_1) \ldots f(x_N)] \in R^N\) and, for a uniform grid spacing of \(\Delta\),

$$\omega \equiv \Delta \begin{bmatrix} \frac{1}{2} & 1 & \ldots & 1 & \frac{1}{2} \end{bmatrix} \in R^N$$

The quadrature rule can be implemented easily as

```
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
out = dot(f.(x), ω)
```

However, in this case the creation of the \(ω\) temporary is inefficient as there are no reasons to allocate an entire vector just to iterate through it with the \(\text{dot}\). Instead, create an iterable by following the interface definition for Iteration, and implement the modified \text{trap_weights} and integration.

Hint: create a type such as

```
struct UniformTrapezoidal
    count::Int
    Δ::Float64
end
```

and then implement the function \text{Base.iterate}(S::UniformTrapezoidal, state=1).

7.9.2 Exercise 1b (Advanced)

Make the \text{UniformTrapezoidal} type operate as an array with interface definition for \text{AbstractArray}. With this, you should be able it go \(ω[2]\) or \(\text{length}(ω)\) to access the quadrature weights.

7.9.3 Exercise 2 (Advanced)

Implement the same features as Exercise 1a and 1b, but for the \text{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

[1]: using InstantiateFromURL
github_project("QuantEcon/quantecon-notebooks-julia", version = "0.5.0")
# github_project("QuantEcon/quantecon-notebooks-julia", version = "0.5.0", instantiate =false) # uncomment to force package installation
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 **QuadGK**.

```plaintext
using QuadGK
@show value, tol = quadgk(cos, -2π, 2π);
```

```
(value, tol) = quadgk(cos, -2π, 2π) = (-1.5474478810961125e-14, 5.7846097329025695e-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) **Gaussian quadrature**.

For example, using **FastGaussQuadrature.jl**

```plaintext
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 **FastGaussQuadrature** package is that you will need to deal with affine transformations to the non-default domains yourself.

Alternatively, **QuantEcon.jl** has routines for Gaussian quadrature that translate the domains.

```plaintext
using QuantEcon
x, w = qnweight(65, -2π, 2π);
@show w * cos.(x); # i.e. on [-2π, 2π] domain
```

```
```
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`.

```julia
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
```

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

```julia
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)
```
To implement linear and cubic spline interpolation

```julia
li = LinearInterpolation(x, y)
lispline = 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, lispline(xf), label = "spline")
```

\[ li(0.3) = 0.25244129544236954 \]
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

```plaintext
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))
```

[9]:

At this point, \texttt{Interpolations.jl} does not have support for cubic splines with irregular grids, but there are plenty of other packages that do (e.g. \texttt{Dierckx.jl} and \texttt{GridInterpolations.jl}).

### 8.4.3 Multivariate Interpolation

Interpolating a regular multivariate function uses the same function

\begin{verbatim}
[f(x,y) = \log(x+y)
x = 1:0.2:5
y = 2:0.1:5
A = [f(x,y) for x in xs, y in ys]

# linear interpolation
interp_linear = LinearInterpolation((xs, ys), A)
@show interp_linear(3, 2) # exactly log(3 + 2)
@show interp_linear(3.1, 2.1) # approximately log(3.1 + 2.1)

# cubic spline interpolation
interp_cubic = CubicSplineInterpolation((xs, ys), A)
@show interp_cubic(3, 2) # exactly log(3 + 2)
@show interp_cubic(3.1, 2.1) # approximately log(3.1 + 2.1);
\end{verbatim}

\begin{verbatim}
interp_linear(3, 2) = 1.6094379124341003
interp_linear(3.1, 2.1) = 1.6484736801441782
interp_cubic(3, 2) = 1.6094379124341
interp_cubic(3.1, 2.1) = 1.6486586594237707
\end{verbatim}

See \texttt{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.
8.6. GENERAL TOOLS

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.

```julia
[11]: using LaTeXStrings
L"an equation: $1 + \alpha^2$"

[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 for loops, etc.

From the documentation

```julia
[12]: using ProgressMeter
@showprogress 1 "Computing..." for i in 1:50
    sleep(0.1) # some computation....
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

```plaintext
[1]: using InstantiateFromURL
    github_project("QuantEcon/quantecon-notebooks-julia", version = "0.5.0")
    # github_project("QuantEcon/quantecon-notebooks-julia", version = "0.5.0", instantiate = false) # uncomment to force package installation

[2]: using LinearAlgebra, Statistics
    using DataFrames, RDatasets, DataFramesMeta, CategoricalArrays, Query, VegaLite
    using DataVoyager, 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

```plaintext
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)
```

<table>
<thead>
<tr>
<th>commod</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>crude</td>
<td>4.2</td>
</tr>
<tr>
<td>gas</td>
<td>11.3</td>
</tr>
<tr>
<td>gold</td>
<td>12.1</td>
</tr>
<tr>
<td>silver</td>
<td></td>
</tr>
</tbody>
</table>

Columns of the `DataFrame` can be accessed by name using `df.col`, as below

```plaintext
df.price
```

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.

```plaintext
df.commod
```

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`.
9.3. DATAFRAMES

[6]:
```
DataFrames.describe(df)
```

<table>
<thead>
<tr>
<th>variable</th>
<th>mean</th>
<th>min</th>
<th>median</th>
<th>max</th>
<th>unique</th>
<th>nmissing</th>
<th>eltype</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol</td>
<td>Union...</td>
<td>Any</td>
<td>Union...</td>
<td>Any</td>
<td>Union...</td>
<td>Type</td>
<td></td>
</tr>
<tr>
<td>1 commod</td>
<td>crude</td>
<td>silver</td>
<td>4</td>
<td>12.1</td>
<td>1</td>
<td>Union{Missing, Float64}</td>
<td></td>
</tr>
</tbody>
</table>

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.

[7]:
```
nt = (commod = "nickel", price = 5.1)
push!(df, nt)
```

```
<table>
<thead>
<tr>
<th>commod</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>Float64</td>
</tr>
<tr>
<td>1 crude</td>
<td>4.2</td>
</tr>
<tr>
<td>2 gas</td>
<td>11.3</td>
</tr>
<tr>
<td>3 gold</td>
<td>12.1</td>
</tr>
<tr>
<td>4 silver</td>
<td></td>
</tr>
<tr>
<td>5 nickel</td>
<td>5.1</td>
</tr>
</tbody>
</table>
```

Named tuples can also be used to construct a DataFrame, and have it properly deduce all types.

[8]:
```
nt = (t = 1, col1 = 3.0)
df2 = DataFrame([nt])
push!(df2, (t = 2, col1 = 4.0))
```

```
MethodError: no method matching eachcolumn(::Tables.CopiedColumns{NamedTuple{(:t, :col1),Tuple{Array{Int64,1},Array{Float64,1}}}})
Closest candidates are:
eachcolumn(!Matched::Union{Function, Type}, !Matched::Tables.Schema{names,nothing}, !Matched::Any) where names at /home/ubuntu/.julia/packages/Tables/TA7NF/src/utils.jl:109
eachcolumn(!Matched::Union{Function, Type}, !Matched::Tables.Schema{names,types}, !Matched::Any) where {names, types} at /home/ubuntu/.julia/packages/Tables/TA7NF/src/utils.jl:66

Stacktrace:
[1] #fromcolumns#410(::Bool, ::typeof(DataFrames.fromcolumns), ::Tables.CopiedColumns{NamedTuple{(:t, :col1),Tuple{Array{Int64,1},Array{Float64,1}}}}) at /home/ubuntu/.julia/packages/DataFrames/yH0f6/src/other/tables.jl:13
[2] (::DataFrames.var"#kw##fromcolumns")::NamedTuple{(:copycols,),Tuple{Bool}} at ./none:0
[3] DataFrame{Array{NamedTuple{(:t, :col1),Tuple{Int64,Float64}},1}} at /home/ubuntu/.julia/packages/DataFrames/yH0f6/src/other/tables.jl:39
[4] top-level scope at In[8]:2
```

In order to modify a column, access the mutating version by the symbol `df[!, :col]`. 
CHAPTER 9. DATA AND STATISTICS PACKAGES

[9]: 

\[
\text{df[:, :price]}
\]

5-element Array\{Union\{Missing, Float64\}, 1\}:
  4.2
  11.3
  12.1
  missing
  5.1

Which allows modifications, like other mutating functions in Julia.

[10]:

\[
\text{df[:, :price] = 2.0} \quad \# \text{double prices}
\]

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.

[11]:

\[
\text{allowmissing!(df2, \text{:col1}) \quad \# \text{necessary to add in a for col1}}
\]

push!(df2, (t=3, \text{col1} = missing))
push!(df2, (t=4, \text{col1} = 5.1))

UndefVarError: df2 not defined

Stacktrace:
[1] top-level scope at In[11]:1

We can see the propagation of missing to caller functions, as well as a way to efficiently calculate with non-missing data.

[12]:

\[
\begin{align*}
\text{@show mean(df2.col1)} \\
\text{@show mean(skipmissing(df2.col1))}
\end{align*}
\]

UndefVarError: df2 not defined
9.3 DATAFRAMES

Stacktrace:
[1] top-level scope at show.jl:562
[2] top-level scope at In[12]:1

And to replace the missing

13: df2.col1 .= coalesce.(df2.col1, 0.0) # replace all missing with 0.0

UndefVarError: df2 not defined

Stacktrace:
[1] top-level scope at In[13]: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.

14: using DataFramesMeta
    f(x) = x^2
    df2 = @transform(df2, col2 = f.(col1))

UndefVarError: df2 not defined

Stacktrace:
[2] top-level scope at In[14]:3

9.3.4 Categorical Data

For data that is categorical

15: using CategoricalArrays
    id = [1, 2, 3, 4]
    y = ["old", "young", "young", "old"]
    y = CategoricalArray(y)
    df = DataFrame(id=id, y=y)

15:
### 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 `|>`.  

```julia
x = 3.0
f(x) = x^2
\[g(x) = \log(x)\]

@show g(f(x))
@show x |> f |> g; # pipes nest function calls
```

\[g(f(x)) = 2.197245773362196\]
\[(x |> f) |> g = 2.197245773362196\]

To give an example directly from the source of the LINQ inspired Query.jl

```julia
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
```
Stacktrace:

[1] #fromcolumns#410(::Bool, ::typeof(DataFrames.fromcolumns), ::Tables.CopiedColumns{NamedTuple{(:name, :children), Tuple{Array{String,1}, Array{Int64,1}}}}) at /home/ubuntu/.julia/packages/DataFrames/yH0f6/src/other/tables.jl:13

[2] (::DataFrames.var"#k#\#fromcolumns")::NamedTuple{(:copycols,), Tuple{Bool}}, ::typeof(DataFrames.fromcolumns), ::Tables.CopiedColumns{NamedTuple{(:name, :children), Tuple{Array{String,1}, Array{Int64,1}}}}) at ./none:0

[3] #DataFrame#412(::Bool, ::Type{DataFrame}, ::QueryOperators.EnumerableMap{NamedTuple{(:name, :children), Tuple{String, Int64}}, QueryOperators.EnumerableFilter{NamedTuple{(:name, :age, :children), Tuple{String, Float64, Int64}}, QueryOperators.EnumerableIterable{NamedTuple{(:name, :age, :children), Tuple{String, Float64, Int64}}, Tables.DataValueRowIterator{NamedTuple{(:name, :age, :children), Tuple{String, Float64, Int64}}, Tables.Schema{(:name, :age, :children), Tuple{String, Float64, Int64}}, Tables.RowIterator{NamedTuple{(:name, :age, :children), Tuple{Array{String,1}, Array{Float64,1}, Array{Int64,1}}}}}}, var"#6#8"}, var"#7#9"})

at /home/ubuntu/.julia/packages/DataFrames/yH0f6/src/other/tables.jl:32

[4] DataFrame(::QueryOperators.EnumerableMap{NamedTuple{(:name, :children), Tuple{String, Int64}}, QueryOperators.EnumerableFilter{NamedTuple{(:name, :age, :children), Tuple{String, Float64, Int64}}, QueryOperators.EnumerableIterable{NamedTuple{(:name, :age, :children), Tuple{String, Float64, Int64}}, Tables.DataValueRowIterator{NamedTuple{(:name, :age, :children), Tuple{String, Float64, Int64}}, Tables.Schema{(:name, :age, :children), Tuple{String, Float64, Int64}}, Tables.RowIterator{NamedTuple{(:name, :age, :children), Tuple{Array{String,1}, Array{Float64,1}, Array{Int64,1}}}}}}, var"#6#8"}, var"#7#9"})

at /home/ubuntu/.julia/packages/DataFrames/yH0f6/src/other/tables.jl:23

[5] top-level scope at In[18]:4

While it is possible to just use the Plots.jl library, there may be better options for displaying tabular data – such as VegaLite.jl.

```julia
using RDatasets, VegaLite
iris = dataset("datasets", "iris")

iris |> @vlplot(ụ:\point, x: PetalLength, y: PetalWidth, color: Species)
```

![Species](setosa, versicolor, virginica)
Another useful tool for exploring tabular data is DataVoyager.jl.

using DataVoyager
iris |> Voyager()

The Voyager() function creates a separate window for analysis.

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.

```
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
```
9.4. STATISTICS AND ECONOMETRICS

[20]: StatsModels.TableRegressionModel{LinearModel{GLM.LmResp{Array{Float64,1}},GLM.DensePredChol{Float64,Cholesky{Float64,Array{Float64,2}}}},Array{Float64,2}}

\[ y \sim 1 + x \]

Coefficients:

\begin{array}{ccccccc}
\hline
\text{Estimate} & \text{Std. Error} & \text{t value} & \text{Pr(>|t|)} & \text{Lower 95%} & \text{Upper 95%} \\
\hline
(\text{Intercept}) & 0.243101 & 0.0156003 & 15.5831 & <1e-27 & 0.212143 & 0.27406 \\
x & 0.917636 & 0.0143706 & 63.8553 & <1e-80 & 0.889118 & 0.946154 \\
\hline
\end{array}

To display the results in a useful tables for LaTeX and the REPL, use \texttt{RegressionTables} for output similar to the Stata package \texttt{esttab} and the R package \texttt{stargazer}.

[21]:

\begin{verbatim}
using RegressionTables
regtable(ols)
# regtable(ols, renderSettings = latexOutput()) # for LaTeX output
\end{verbatim}

\begin{verbatim}
----------------------
y
--------
(1)
----------------------
(Intercept) 0.243***
(0.016)
x 0.918***
(0.014)
----------------------
Estimator OLS
----------------------
N 100
R2 0.977
----------------------

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 \texttt{cigarette consumption data}

[22]:

\begin{verbatim}
using FixedEffectModels
cigar = dataset("plm", "Cigar")
cigar.StateCategorical = categorical(cigar.State)
cigar.YearCategorical = categorical(cigar.Year)
fixedeffectresults = reg(cigar, @model(Sales ~ NDI, fe = StateCategorical + YearCategorical, weights = Pop, vcov = cluster(StateCategorical)))
regtable(fixedeffectresults)
\end{verbatim}

\begin{verbatim}
----------------------
Sales
--------
(1)
----------------------
NDI -0.005***
\end{verbatim}
To explore data use the interactive DataVoyager and VegaLite.

```julia
[23]:
cigar = dataset("plm", "Cigar")
# cigar |> Voyager()

cigar |> @vlplot(
  :point,
  x=:Price,
  y=:Sales,
  color=:Year,
  size=:NDI
)
```

```text
TypeError: Cannot read property 'getContext' of null
at resize (/home/ubuntu/.julia/packages/VegaLite/sHyyT/deps/node_modules/vega-scenegraph/build/vega-scenegraph.js:3377:26)
at CanvasRenderer.prototype$6.resize (/home/ubuntu/.julia/packages/VegaLite/sHyyT/deps/node_modules/vega-scenegraph/build/vega-scenegraph.js:3427:5)
at CanvasRenderer.prototype$6.initialize
```
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

```
[1]: using InstantiateFromURL
   github_project("QuantEcon/quantecon-notebooks-julia", version = "0.5.0")
   # github_project("QuantEcon/quantecon-notebooks-julia", version = "0.5.0", instantiate = true) # uncomment to force package installation

[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 \( R^N \rightarrow R \) 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 \( R^N \rightarrow R \), 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 \( R \rightarrow R^N \), 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 \( \frac{dy}{dw} \) 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.

```plaintext
[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
end

return w_5
[end]
```

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 \( \frac{\partial w_4}{\partial x_1} = 0 \).

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

$$
\begin{align*}
(x + x'\epsilon) + (y + y'\epsilon) &= (x + y) + (x' + y')\epsilon \\
(x + x'\epsilon) \times (y + y'\epsilon) &= (xy) + (x'y + y'x)\epsilon \\
\exp(x + x'\epsilon) &= \exp(x) + (x' \exp(x))\epsilon
\end{align*}
$$

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 then we would seed them with the dual numbers $x_1 \rightarrow (3.8, 1)$ and $x_2 \rightarrow (6.9, 0)$.

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.
10.3. INTRODUCTION TO DIFFERENTIABLE PROGRAMMING

using ForwardDiff

\[ h(x) = \sin(x[1]) + x[1] \cdot x[2] + \sinh(x[1] \cdot x[2]) \] # 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) = sum(sin, x) + prod(tan, x) * sum(sqrt, x)

g = (x) -> ForwardDiff.gradient(f, x); # g() is now the gradient

g(rand(5)) # gradient at a random point

# ForwardDiff.hessian(f, x') # or the hessian

ForwardDiff.gradient(h, x) = [26.354764961030977 16.663053156992284]

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)

1.4142135623730951

using ForwardDiff
dsqr = ForwardDiff.derivative(squareroot, x)
dsqr(2.0)

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.

using Zygote

\[ h(x, y) = 3x^2 + 2x + 1 + y'x - y \]

gradient(h, 3.0, 5.0)

(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.,

\[
D(f) = x \to \text{gradient}(f, x)[1] \quad \# \text{return first in tuple}
\]

\[
D_{\sin}(\sin) = D_{\sin}(4.0)
\]

-0.6536436208636119

For functions of one (Julia) variable, we can find the by simply using the \('\) after a function name

\[
\text{using Statistics}
\]

\[
p(x) = \text{mean(abs, x)}
\]

\[
p'(\{1.0, 3.0, -2.0\})
\]

3-element Array{Float64,1}:

0.3333333333333333
0.3333333333333333
-0.3333333333333333

Or, using the complicated iterative function we defined for the squareroot,

\[
\text{squareroot}'(2.0)
\]

0.3535533905932737

You can Zygote supports combinations of vectors and scalars as the function parameters.

\[
h(x, n) = (\text{sum}(x.^n))^{(1/n)}
\]

\[
\text{gradient}(h, [1.0, 4.0, 6.0], 2.0)
\]

([0.13736056394868904, 0.5494422557947561, 0.8241633836921343],
-1.2725553130925444)

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.

\[
\text{using Optim, LinearAlgebra}
\]

\[
N = 1000000
\]

\[
y = \text{rand}(N)
\]

\[
\lambda = 0.01
\]

\[
\text{obj}(x) = \text{sum}((x\cdot y)^2) + \lambda\cdot\text{norm}(x)
\]

\[
x_{\text{iv}} = \text{rand}(N)
\]

\[
\text{function g!(G, x)}
\]

\[
G = \text{obj}'(x)
\]

\[
\text{end}
\]

\[
\text{results} = \text{optimize} \text{obj}, g!, x_{\text{iv}}, \text{LBFGS()}) \quad \# \text{or ConjugateGradient()}
\]

\[
\text{println("minimum = $\{\text{results.min}\} \text{ with in "}
\]

\[
\text{"$\{\text{results.iterations}\} \text{ iterations"})
\]

minimum = 5.773482174458855 with in 2 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.

```julia
using Optim
using Optim: converged, maximum, maximizer, minimizer, iterations #some extra functions
result = optimize(x -> x^2, -2.0, 1.0)
```

#### Results of Optimization Algorithm

- Algorithm: Brent’s Method
- Search Interval: [-2.000000, 1.000000]
- 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

```julia
converged(result) || error("Failed to converge in \$\text{iterations(result)} iterations")
xmin = result.minimizer
result.minimum
```
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
f(x) = -x^2
result = maximize(f, -2.0, 1.0)
converged(result) || error("Failed to converge in $\text{iterations}(result)$ iterations")
xmin = maximizer(result)
fmax = maximum(result)
```

```
results = optimize(f, x_iv) # i.e. optimize(f, x_iv, NelderMead())
```

### Unconstrained Multivariate Optimization

There are a variety of algorithms and options for multivariate optimization.

From the documentation, the simplest version is

```python
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())
```

```
* 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
    * $\sqrt{\sum(y - y')^2}/n \leq 1.0e-08$
  * 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

```python
results = optimize(f, x_iv, LBFGS())
```

```
minimum = 5.37840461498115e-17 with argmin = [0.9999999926662393, 0.9999999853324786] in 24 iterations
```
10.4. OPTIMIZATION

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.

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

For derivative-free methods, you can change the algorithm – and have no need to provide a gradient

* Status: failure (reached maximum number of iterations) (line search failed)

* Candidate solution
  Minimizer: [9.93e-01, 9.80e-01]
  Minimum: 4.357348e-03

* Found with Algorithm: Simulated Annealing
  Initial Point: [0.00e+00, 0.00e+00]

* Convergence measures
  $|x - x'| = NaN ≈ 0.0e+00$
  $|x - x'|/|x'| = NaN ≈ 0.0e+00$
  $|f(x) - f(x')| = NaN ≈ 0.0e+00$
  $|f(x) - f(x')|/|f(x')| = NaN ≈ 0.0e+00$
  $|g(x)| = NaN ≈ 1.0e-08$

* Work counters
  Seconds run: 0 (vs limit Inf)
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.

```plaintext
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)
```

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 | 0.00e+00 | 0
  1 | -1.4100714e+00 | 0.00e+00 | 5.48e-02 | -1.7 | 3.94e-01 | -1.00e+00 | 7.36e-01f | 1
  2 | -1.4113851e+00 | 0.00e+00 | 2.83e-08 | -2.5 | 9.29e-04 | -1.00e+00 | 1.00e+00f | 1
  3 | -1.4140632e+00 | 0.00e+00 | 1.50e-03 | -3.8 | 1.89e-03 | -1.00e+00 | 1.00e+00f | 1
  4 | -1.4142117e+00 | 0.00e+00 | 1.84e-11 | -5.7 | 1.05e-04 | -1.00e+00 | 1.00e+00f | 1
  5 | -1.4142136e+00 | 0.00e+00 | 8.23e-09 | -8.6 | 1.30e-06 | -1.00e+00 | 1.00e+00f | 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) = 1.557
Total CPU secs in NLP function evaluations = 1.420

EXIT: Optimal Solution Found.
JuMP.optimize!(m) = nothing

And this is an example of a quadratic objective

```
# solve
# min (1-x)^2 + 100(y-x^2)^2)
# s.t 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))
```

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
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  0.00e+00  0 0
1  9.5312500e-01  1.25e+01  1.00e+00  -1.0  2.50e-01f  3
2  4.3820569e-01  1.01e+00  9.03e-02  -1.0  5.00e-01f  1
3  4.5708292e-01  9.53e+00  4.29e-01  -1.0  1.00e+00f  2
4  1.8894265e-01  4.15e-01  9.51e-02  -1.0  1.00e+00f  1
5  1.3918726e-01  6.51e+00  3.49e-01  -1.0  1.00e+00f  2
6  5.4940090e-02  4.51e-01  9.29e-02  -1.0  1.00e+00f  1
7  2.9144630e-02  2.74e+00  1.00e+00  -1.0  1.00e+00f  1
8  9.5856451e-03  1.15e+00  1.00e+00  -1.0  1.00e+00f  1
9  2.3237475e-03  1.00e+00  1.00e+00  -1.0  1.00e+00f  1

Objective: 1.3288608467480825e-28
Dual infeasibility: 2.0183854587685121e-13
Constraint violation: 0.0000000000000000e+00
Complementarity: 0.0000000000000000e+00
Overall NLP error: 2.0183854587685121e-13

EXIT: Optimal Solution Found.

x = 0.9999999999999999
y = 0.9999999999999792

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

iter  objective   inf_pr   inf_du   lg(mu)   ||d||   lg(rg)   alpha_du   alpha_pr   ls
0  1.0000000e+00  0.00e+00  1.00e+00  -1.0  0.00e+00  -0.00e+00  0.00e+00  0 0
1  9.6312500e-01  1.25e+01  1.00e+00  -1.0  2.50e-01f  3
2  1.6901463e-01  8.31e-04  2.50e-03  -1.0  1.00e+00f  1
3  1.0915040e-01  8.68e-07  5.70e-05  -1.0  1.00e+00f  1
4  1.3288608e-28  2.02e-03  8.45e-08  -1.0  1.00e+00f  1

14
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 Lagrangian Hessian evaluations = 14
Total CPU secs in IPOPT (w/o function evaluations) = 0.078
Total CPU secs in NLP function evaluations = 0.011

EXIT: Optimal Solution Found.

x = 0.9999999999999999
y = 0.9999999999999792

This is Ipopt version 3.12.10, running with linear solver mumps.
NOTE: Other linear solvers might be more efficient (see Ipopt documentation).
10.4. Optimization

<table>
<thead>
<tr>
<th></th>
<th>2.543317e+04 1.70e-15 3.18e+04</th>
<th>-1.0 2.05e+00</th>
<th>1.00e+00 1.00e+00f</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2.652775e+03 0.00e+00 7.79e+03</td>
<td>-1.0 1.19e+00</td>
<td>1.00e+00 1.00e+00f</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1.138032e+02 0.00e+00 1.35e+03</td>
<td>-1.0 5.62e-01</td>
<td>1.00e+00 1.00e+00f</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>3.374550e+00 0.00e+00 8.45e+01</td>
<td>-1.0 1.06e+01</td>
<td>1.00e+00 1.00e+00f</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2.894619e+00 0.00e+00 4.22e-01</td>
<td>-1.0 1.07e-02</td>
<td>1.00e+00 1.00e+00f</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>2.894607e+00 0.00e+00 1.07e-05</td>
<td>-1.7 5.42e-05</td>
<td>1.00e+00 1.00e+00f</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>2.894607e+00 0.00e+00 5.91e-13</td>
<td>-8.6 1.38e-09</td>
<td>1.00e+00 1.00e+00f</td>
<td></td>
</tr>
</tbody>
</table>

Number of Iterations...: 9

<table>
<thead>
<tr>
<th>(scaled)</th>
<th>(unscaled)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective...: 2.8946075504894599e+00</td>
<td>2.8946075504894599e+00</td>
</tr>
<tr>
<td>Dual infeasibility...: 5.9130478291535837e-13</td>
<td>5.9130478291535837e-13</td>
</tr>
<tr>
<td>Complementarity...: 0.0000000000000000e+00</td>
<td>0.0000000000000000e+00</td>
</tr>
<tr>
<td>Overall NLP error...: 5.9130478291535837e-13</td>
<td>5.9130478291535837e-13</td>
</tr>
</tbody>
</table>

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) = 0.001
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

```julia
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}}
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 = 111895.40
Function evals per second = 113070.18
Improvements/step = 0.19380
Total function evaluations = 10106

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

```julia
using Roots
f(x) = sin(4 * (x - 1/4)) + x + x^{20} - 1
fzero(f, 0, 1)
```

[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

```julia
using NLsolve
f(x) = [(x[1] + 3)*(x[2]^{3} - 7) + 18
  sin(x[2] * exp(x[1] - 1))]
# returns an array
```

[25]:
results = nlsolve(f, [ 0.1; 1.2])

Results of Nonlinear Solver Algorithm
* Algorithm: Trust-region with dogleg and autoscaling
* Starting Point: [0.1, 1.2]
* Zero: [-7.775508345910301e-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.

```
results = nlsolve(f, [ 0.1; 1.2], autodiff=:forward)
println("converged=$\{(NLsolve.converged(results))\} at root=$\{(results.zero)\} in """
"\$(results.iterations) iterations and \$(results.f_calls) function calls")
```

```
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)\} in """
"\$(results.iterations) iterations and \$(results.f_calls) function calls")
```

```
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)\} in """
"\$(results.iterations) iterations and \$(results.f_calls) 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).

10.6 LeastSquaresOptim.jl

Many optimization problems can be solved using linear or nonlinear least squares.

Let \( x \in \mathbb{R}^N \) and \( F(x) : \mathbb{R}^N \to \mathbb{R}^M \) with \( M \geq N \), then the nonlinear least squares problem is

\[
\min_x F(x)^T F(x)
\]

While \( F(x)^T F(x) \to R \), 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:

```plaintext
using LeastSquaresOptim
function rosenbrock(x)
    [1 - x[1], 100 * (x[2] - x[1]^2)]
end
LeastSquaresOptim.optimize(rosenbrock, zeros(2), Dogleg())
```

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.

```plaintext
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))
```

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.

```
[30]:
struct DualNumber{T} <: Real
    val::T
    ￿::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

```
[31]:
import Base: +, *, -, ^, exp
+{x::DualNumber, y::DualNumber} = DualNumber(x.val + y.val, x.￿ + y.￿) # dual addition
+{x::DualNumber, a::Number} = DualNumber(x.val + a, x.￿) # i.e. scalar addition, not dual
+{a::Number, x::DualNumber} = DualNumber(x.val + a, x.￿) # i.e. scalar addition, not dual
```

With that, we can seed a dual number and find simple derivatives,

```
[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)
```

```
[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.