Course Overview and Computational Environment

Undergraduate Computational Macro

Jesse Perla

jesse.perla@ubc.ca

University of British Columbia

Table of contents

Course Overview and Objectives

Course Structure and Prerequisites

- “Macroconomics on a computer”. Mostly macro-finance and macro-labor
  - → Not an intro to programming course or stats/econometrics class
  - → Less programming than ECON323, more math and theory
- Build experience with computational tools and structural models in macroeconomics which can help you conduct “counterfactuals”
  - → Lots of simulation, but not much data or empirics
  - → Complement to other courses focusing on “field” topics, empirics, estimation, inference, datascience, etc.

Prerequisites

- You need to have
  - → One of ECON 301, ECON 304, ECON 308
  - → One of ECON 323, CPSC 103, CPSC 110, MATH 210, COMM 337
  - → One of MATH 221, MATH 223.
- **Not negotiable** to have intermediate micro
- **Not negotiable** to have the formal programming class in some general purpose language (e.g., Stata and R don't count, self-study isn't enough)
- Math requirement you can talk to me, especially if you took ECON307 or have significant background in linear algebra and multivariate calculus

Assessments

- Grading:
  - → 6-8 problem sets: 20% (total)
  - → Midterm exam: 30%
  - → Final exam: 50%
- Midterm and final examinations will be done in a computer lab or on your own computer in class. Not testing programming skills
- Problem sets will start off short and easy to help those with less programming experience, and then build in (economics) complexity.
- See the syllabus for missed exam policies

Programming Languages

Which Language?

- Plenty of languages used in economics and finance: Matlab, Python, Julia, Fortran, C++, Stata, Dynare, R, Stan...
  - → All are great for some things, and terrible for others
  - → Some are highly specialized and less general purpose than others (e.g. Stata and R)
- I love specialized languages! But...
  - → My philosophy is you will need to learn **at least** two general purpose programming languages over your career.

Benefits of Learning more Languages

Plan for your longrun career, languages come and go...

- The 2nd language makes you a better programmer at both
- The 3rd is even easier as you learn similarities and differences
- On grad school or job applications everyone says they know Python
  - → Differentiator to credibly claim you know another serious language
  - → Increasingly important to **signal** computational sophistication to get jobs
  - → Julia is as good as any for that purpose

Advantages of Learning Julia for Economics and Finance

- Python is great for datascience and ML, but “ugly”, verbose, and slow to use directly for many simulations and computational methods
  - → Python wrappers for high-performance code used in ML are great
  - → But when an appropriate framework doesn’t exist, writing fast code yourself in Python is much harder than in Julia
  - → Performance in Python usually means C++ or frameworks like JAX
- Julia (and Matlab) is more natural for programming mathematics than Python. Easier to learn than alternative Python packages.
- Many in economists and finance research use Julia for computational methods, so it may help you directly

Don't Worry If You are New to Programming

- Costs of learning languages has decreasing returns to scale
  - → Learning the **first programming language is the hardest**
- Julia will come easily if you have the prerequisities (i.e. a course using Matlab or Python, sadly R is not sufficient preparation)
- Submitting your code in Matlab or Python is not possible given the course structure and infrastructure

Quantitative, Empirical, and Theoretical Economics

Why Isn't Big Data ML/Statistics Enough?

- Going well before the big data/ML revolution economists asked whether they could just use statistical models with enough data
  - → Answer: only if you had the right (statistical) model for a particular experiment, but historical data doesn't have variation in crucial directions
  - → The right "statistical model" would need to reflect that humans adapt and make forecasts - responding to policy and incentives
  - → Especially difficult in macro because of dynamics and GE effects
  - → Cowles Commision, Lucas Critique, Policy Ineffectiveness Proposition (Sargent and Wallace), Time Inconsistency (Kydland and Prescott)
- Having more data and fancier statistics doesn't solve these problems

Forecasts and Distributions

- Summary: conducting experiments with a data generating process (DGP) is fine, but how to find the right one **for a given problem**?
- Think probabilistically: the world is a joint distribution of observables, unobservables (i.e., latent variables), shocks, and parameters
- Joint distributions let you calculate conditional expectations and conduct “experiments” by conditioning on different events
- Statistics and machine learning is often criticized as being only about “prediction” and sometimes “inference”
  - → This isn’t quite true, but lets us ask what prediction really means

Counterfactuals: “What If?”

- Most interesting problems in economics are about counterfactuals
  - → What would unemployment have been if the government had not intervened during the recession?
  - → What would have been her income if she had not gone to college, or if she wasn’t subjected to gender bias?
- By definition these are not observable. If we had the data already we wouldn’t need to ponder these “What if?”
- How can you answer a question with data that doesn’t exist?

YOU HAVE TO MAKE SOMETHING UP

The Role of Theory

- There is no data interpretation without some theory - even if it is sometimes implicit. Interpreting empirical results require self-reflection
- The role of both data and theory is then to help constrain the set of possible counterfactuals for the "what if?"
- So any criticisms of ML or statistics as "merely prediction" are basically a statement on whether the theory makes sense
  - → i.e., if you fit $y = f(X) + \epsilon$ on data to find a $\hat{f}(X)$ function, then theory tells you if you made the right assumptions (e.g., that the X data is representative and wouldn't change for your counterfactual of interest, etc)

Approach in this Course

- Always remember: you need assumptions in one form or another because the counterfactuals are inherently not in the data
- Broadly there are three approaches to conducting counterfactuals. They are not mutually exclusive
    1. Structural models emphasize theory as structure on the joint distribution
    2. Causal inference using matching, instrumental variables, etc. which use theoretical assumptions on independence to adjust for bias and missing unobservable (latent) variables
    3. Randomized Experiments/Treatment Effects where you can get good data which truly randomizes some sort of “treatment”.
- In this course we will focus on **simulations and structural models** - sometimes called “quantitative economics”

Macroeconomic Models Require Lots of Tools

- Conducting macroeconomic counterfactuals requires a lot of tools because
  - → Macroeconomic decisions are dynamic and often stochastic
  - → Agents are forward looking
  - → Agents interact through markets and prices, which creates “general equilibrium” effects (i.e., which are inherently nonlinear)
  - → Heterogeneity leads to the distributional being crucial
  - → Agent’s may respond to policies by thinking through the dynamic effects
- We formalize these assumptions with math, but we are rarely able to solve them analytically. Use a computer!

Tools Topics

See Syllabus for more details

1. Linear algebra and basic scientific computing
2. Geometric Series and Discrete Time Dynamics
3. Basic Stochastic Processes
4. Linear State Space Models
5. Markov Chains
6. Dynamic Programming

Applications Topics

The tools are interleaved with applications such as

1. Marginal Propensity to Consume
2. Dynamics of Wealth and Distributions
3. Permanent Income Model
4. Models of Unemployment
5. Asset Pricing
6. Lucas Trees and No-arbitrage Option Pricing
7. Recursive Equilibria and the McCall Search Model
8. Time permitting: Rational Expectations and Firm Equilibria, Growth Models

Computational Environment

Setup

- You can install Julia on your laptop by following **these instructions**
- While one can use Julia entirely from just Jupyter notebooks, we will also introduce basic **GitHub** and **VS Code** usage as well to help broaden your exposure to computational tools.
- So my suggestion is to challenge yourself to learn VS Code, GitHub, and other tools. Further **signalling** for RA/predoc/jobs/etc.

Summary of Installation

1. Install **Git**
2. Install **Anaconda**
3. Install Julia with **juliaup**
    - Windows: easiest method is `winget install julia -s msstore` in a Windows terminal
    - Linux/Mac: in a terminal use `curl -fsSL https://install.julialang.org | sh`
4. Install **Visual Studio Code (VS Code)**
5. Install the **VS Code Julia** extension

Some Common Errors on MacOS

- To open a terminal on MacOS
  - → Press `Cmd + Space` to open Spotlight, then type `Terminal`
  - → Or with VS Code `<Cmd-Shift-P>` then `View: Toggle Terminal`
- If you get permissions problems try

```
sudo curl -fsSL https://install.julialang.org | sh
```

- If it still shows errors, then see here and do some combo of

```
sudo chown $(id -u):$(id -g) ~/.bashrc
sudo chown $(id -u):$(id -g) ~/.zshrc
sudo chown $(id -u):$(id -g) ~/.bash_profile
```

  - → Then retry `sudo curl -fsSL https://install.julialang.org | sh`

Clone Notebooks and Install Packages

1. Open the command palette with `<Ctrl+Shift+P>` or `<Cmd+Shift+P>` on mac and type `> Git: Clone` and choose `https://github.com/jlperla/undergrad_computational_macro_notebooks`
2. Instantiate packages, in VSCode or
   - Run a terminal in that directory
   - Then `julia` and `]` enters package mode
   - `] add IJulia`, which adds to global environment
   - `] activate`, which chooses the `Project.toml` file
   - `] instantiate`
3. Then use VS Code or `jupyter lab` to open

Julia Environment Basics

- Project files keep track of dependencies and make things reproducible
  - → Similar to Python's virtual environments but easier to use
- VS Code and Jupyter will automatically activate a `Project.toml`
  - → In REPL or Jupyter enter `]` for managing packages
  - → Can manually activate with `] activate` or `] activate path/to/project`
  - → On commandline, can use `julia --project`
  - → If a file doesn't exist, then `]activate` creates one for the folder
- With activated project, use `] instantiate` to install all the packages
- For this course: no package management required after instantiation

Reproducibility

- **ALWAYS** use a `Project.toml` file
  - → Keep your global environment as clean
  - → Enough to do `] add IJulia`
- Associated with `Project.toml` is a `Manifest.toml` file which establishes the exact versions for reproducibility
  - → `] instantiate` will install the exact versions
  - → Less important for us, but very useful for reproducibility in research to distribute with project

Crash Course on Julia

Introductory Lectures

- Assuming you are familiar with Matlab or Python, Julia will be easy to learn
- Adapted from QuantEcon lectures coauthored with John Stachurski and Thomas J. Sargent
  - → Julia by Example
  - → Essentials
  - → Fundamental Types

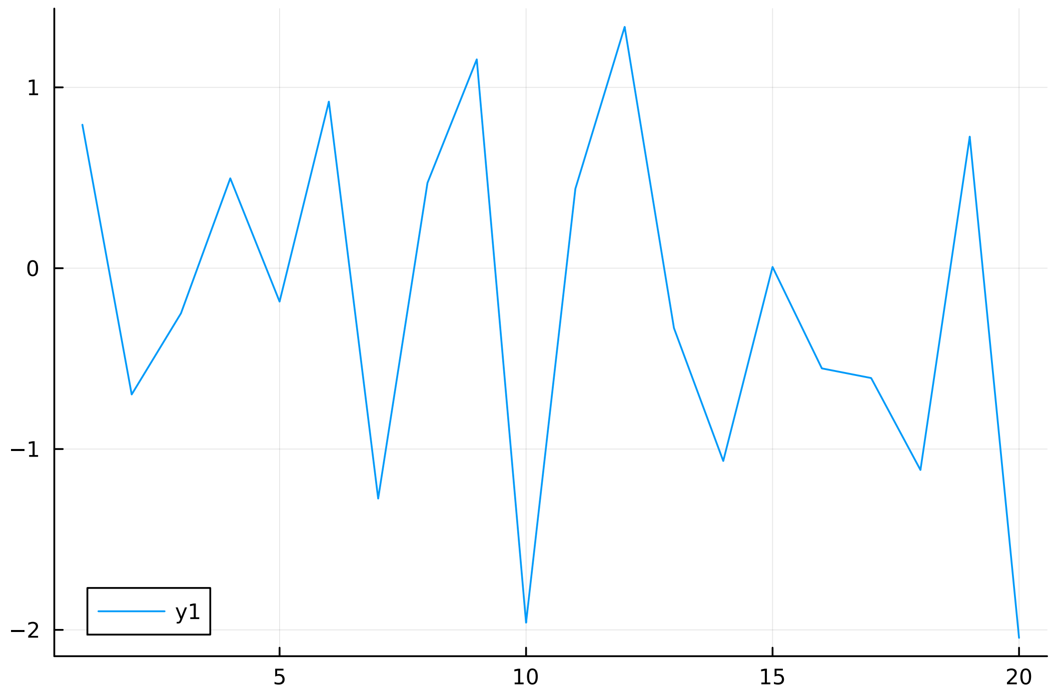
Using Packages

- First ensure your project is activated and packages instantiated

```julia
using LinearAlgebra, Statistics, Plots
```

Plotting Random Numbers

```
n = 20
ep = randn(n)
plot(1:n, ep;size=(600,400))
```



Loops

```
n = 100
ep = zeros(n)
for i in 1:n
    ep[i] = randn()
end
println(ep[1:5])
```

```
[-1.064821744918741, 0.200553208140404 25, -0.42053012088019653, -2.1674797424122554, -0.9601569259178233]
```

Comprehensions

```julia
# Comprehensions
@show [2 * i for i in 1:4];
```

```
[2i for i = 1:4] = [2, 4, 6, 8]
```

Manually Calculated Mean

```
ep_sum = 0.0 # careful to use 0.0 here, instead of 0
for ep_val in ep
    ep_sum = ep_sum + ep_val
end
@show ep_mean = ep_sum / length(ep)
@show ep_mean ≈ mean(ep)
@show ep_mean
@show sum(ep) / length(ep)
@show sum(ep_val for ep_val in ep) / length(ep); # generator/comprehension
```

```
ep_mean = ep_sum / length(ep) = -0.014019546438837875
ep_mean ≈ mean(ep) = true
ep_mean = -0.014019546438837875
sum(ep) / length(ep) = -0.014019546438837903
sum((ep_val for ep_val = ep)) / length(ep) = -0.014019546438837875
```

Functions

```
function generatedata(n)
    ep = randn(n) # use built in function
    for i in eachindex(ep) # or i in 1:length(ep)
        ep[i] = ep[i]^2 # squaring the result
    end
    return ep
end
data = generatedata(5)
println(data)
```

```
[1.7182747971605918, 0.017624557346776663, 1.0111342207535723, 3.2936289192315935, 0.6153258611237733]
```

Broadcasting

```julia
function generatedata(n)
    ep = randn(n) # use built in function
    return ep .^ 2
end
@show generatedata(5)
generatedata2(n) = randn(n) .^ 2
@show generatedata2(5);
```

```
generatedata(5) = [0.7664969548681376, 0.5658795847535621, 0.1920865182464282, 0.44516414349150646, 0.4335964686270287]
generatedata2(5) = [3.231571451656551, 0.26727719282014856, 5.756213117912331, 0.32414937784829506,
0.008613087034908314]
```

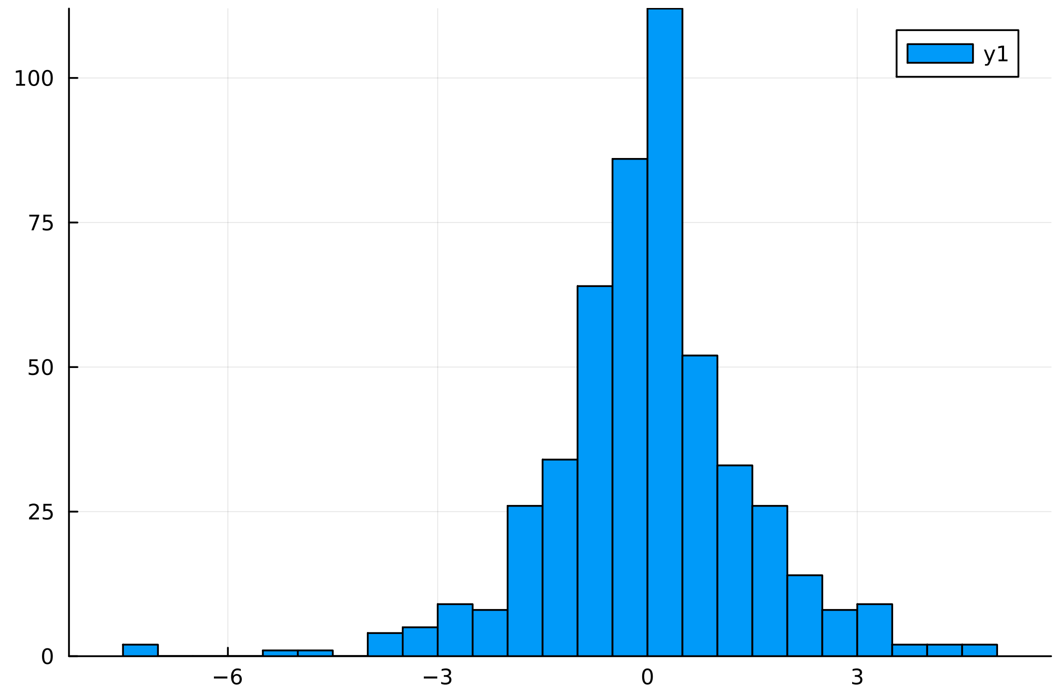
Higher Order Functions

```
generatedata3(n, gen) = gen.(randn(n)) # broadcasts on gen
f(x) = x^2 # simple square function
@show generatedata3(5, f); # applies f
```

```
generatedata3(5, f) = [9.712146802111512, 0.3200963448050346, 1.1942889536301105, 0.07268419480565189, 0.001438483083349692]
```

More Plotting Examples

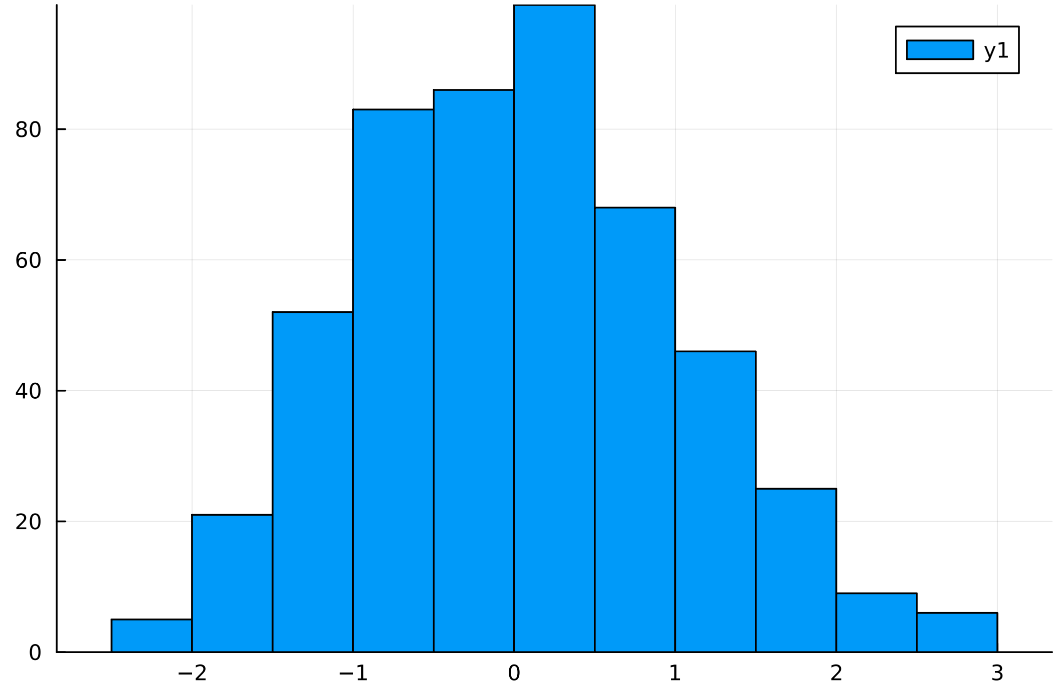
```julia
using Distributions
function plothistogram(dist, n)
    # n draws from distribution
    ep = rand(dist, n)
    return histogram(ep;size=(600,400))
end
dist = Laplace() # dist != dist in function
plothistogram(dist, 500)
```



Changing Types

- The `rand(dist, n)` changes its behavior based on the type of `dist`

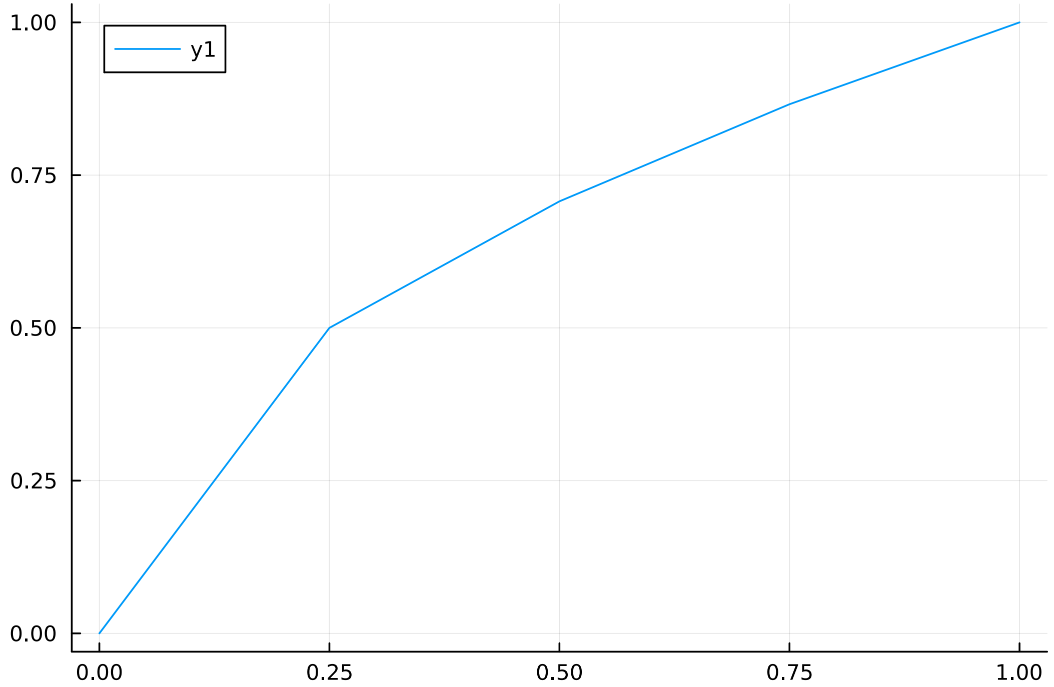
```
dist = Normal()
plothistogram(Normal(), 500)
```



Ranges

```
x = range(0.0, 1.0; length = 5)
@show x
@show Vector(x)
plot(x, sqrt.(x);size=(600,400))
```

```
x = 0.0:0.25:1.0
Vector(x) = [0.0, 0.25, 0.5, 0.75, 1.0]
```



Defining Functions

- You can create anonymous functions as in R, but it is harder for the compiler because the type `f3` can change. Avoid `->` if name required

```
f(x) = x^2
function f2(x)
    return x^2
end
f3 = x -> x^2 # assignment not required
@show f(2), f2(2), f3(2);
```

```
(f(2), f2(2), f3(2)) = (4, 4, 4)
```

Default Arguments

```julia
f(x, a = 1) = exp(cos(a * x))
@show f(pi)
@show f(pi, 2);
```

```
f(pi) = 0.36787944117144233
f(pi, 2) = 2.718281828459045
```

Keyword Arguments

```
f2(x; a = 1) = exp(cos(a * x))  # note the ; in the definition
# same as longform
function f(x; a = 1)
    return exp(cos(a * x))
end
@show f(pi)
@show f(pi; a = 2) # passing in adate
a = 2
@show f(pi; a); # equivalent to f(pi; a = a)
```

```
f(pi) = 0.36787944117144233
f(pi; a = 2) = 2.718281828459045
f(pi; a) = 2.718281828459045
```

Closures

- In general, try to avoid globals and closures outside of functions

```julia
a = 0.2
f(x) = a * x^2  # refers to the `a` in the outer scope
@show f(1)
# The a is captured in this scope by name.  Careful!
a = 0.3
@show f(1);
```

```
f(1) = 0.2
f(1) = 0.3
```

Closures Inside Functions

- But within a function they are safe, common, and usually free of overhead

```
function g(a)
    f(x) = a * x^2  # refers to the `a` passed in the function
    return f(1)
end
a = 123.5 # Different scope than the `a` in function
@show g(0.2);
```

```
g(0.2) = 0.2
```

Tuples and Named Tuples

```julia
t = (1, 2.0, "hello")
@show t[1]
nt = (;a = 1, b = 2.0, c = "hello")
@show nt
@show nt.a; # can't use nt[1] or nt["a"]
```

```
t[1] = 1
nt = (a = 1, b = 2.0, c = "hello")
nt.a = 1
```

Tuples Packing and Unpacking

```julia
function solve_model(x)
    a = x^2
    b = 2 * a
    c = a + b
    return (; a, b, c)  # note local scope of tuples!
end
@show solve_model(0.1)
# can unpack in different order, or use subset of values
(; c, a) = solve_model(0.1)
println("a = $a, c = $c");
```

```
solve_model(0.1) = (a = 0.010000000000000002, b = 0.020000000000000004, c = 0.030000000000000006)
a = 0.010000000000000002, c = 0.030000000000000006
```

Array Basics

```julia
b = [1.0, 2.1, 3.0] # 1d array
A = [1 2; 3 4] # 2x2 matrix
@show size(b)
@show size(A)
@show typeof(b)
@show typeof(A)
@show zeros(3)
@show ones(2, 2)
@show fill(1.0, 2, 2)
@show similar(A)
@show A[1, 1]
@show A[1, :]
@show A[1:end, 1];
```

```
size(b) = (3,)
size(A) = (2, 2)
typeof(b) = Vector{Float64}
typeof(A) = Matrix{Int64}
zeros(3) = [0.0, 0.0, 0.0]
ones(2, 2) = [1.0 1.0; 1.0 1.0]
fill(1.0, 2, 2) = [1.0 1.0; 1.0 1.0]
similar(A) = [0 0; 0 0]
A[1, 1] = 1
A[1, :] = [1, 2]
A[1:end, 1] = [1, 3]
```

Linear Algebra Basics

```julia
A = [1 2; 3 4]
b = [1, 2]
@show A * b # Matrix product
@show A' # transpose
@show dot(b, [5.0, 2.0]) # dot product
@show b' * b # dot product
@show Diagonal([1.0, 2.0]) # diagonal matrix
@show I # identity matrix
@show inv(A); # inverse
```

```
A * b = [5, 11]
A' = [1 3; 2 4]
dot(b, [5.0, 2.0]) = 9.0
b' * b = 5
Diagonal([1.0, 2.0]) = [1.0 0.0; 0.0 2.0]
I = UniformScaling{Bool}(true)
inv(A) = [-1.9999999999999996 0.9999999999999998;
1.4999999999999998 -0.4999999999999999]
```

Modifying Vectors

- Scalars and tuples/named tuples are immutable
- Vectors and matrices are mutable

```julia
A = [1 2; 3 4]
A[1, 1] = 2
@show A
b = [1, 2]
b[1] = 2
@show b
b .= [3, 4] # otherwise just renamed
@show b
A[1, :] .= [3, 4] # assign slice
@show A;
```

```
A = [2 2; 3 4]
b = [2, 2]
b = [3, 4]
A = [3 4; 3 4]
```

Learning More

- Clone the QuantEcon lectures
  - → `> Git: Clone` the `https://github.com/quantecon/lecture-julia.notebooks`
- This covers part of **Julia Essentials** and **Fundamental Types**
- Other more advanced lectures, not required for this course, are
  - → **Introduction to Types and Generic Programming**
  - → **Generic Programming**
  - → **Visual Studio and Other Tools**