Introduction to Julia

Course 1 — Short Macroeconomics Course Using Julia

November 25, 2025

Outline

1. Julia in a Nutshell
2. Packages and Environments
3. Variables and Types
4. Arrays and Matrices
5. Functions
6. Control Flows
7. Data and Dates
8. Plotting
9. Advanced Concepts
10. Wrap-Up

Course Goals

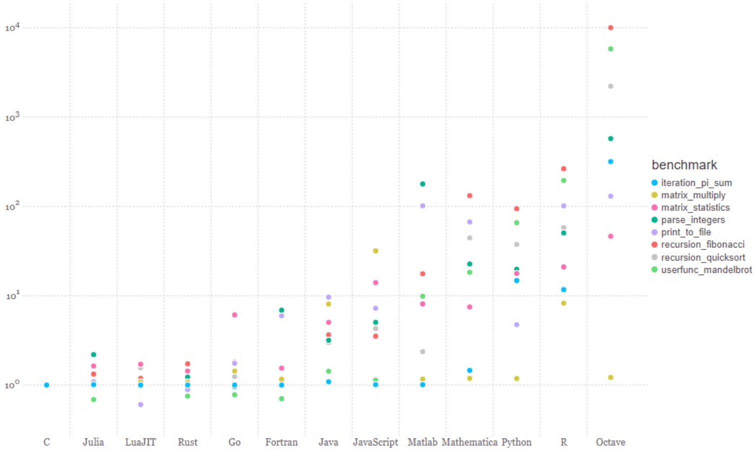
- Understand what Julia is and why it matters for macro modelling.
- Get comfortable with Julia's REPL, packages, and basic syntax.
- Be able to manipulate arrays, write functions, and control flow.
- Load/save data, handle dates, and make basic plots.
- Glimpse advanced features: types, multiple dispatch, vectorization.

Why Julia?

- Julia was designed for high performance.
- Main appeal: C/Fortran speed with Matlab-like syntax.
- Dynamically typed.
- Julia uses multiple dispatch as a paradigm.
- Open-source.
- TIOBE index: https://www.tiobe.com/tiobe-index/, in the top 30.
- State of Julia: https://juliacon.org/2025/

Why Julia?

Figure: Comparison of Julia Speed



Macro Modelling in Julia

- General reference: [Stachurski and Sargent(2017)], https://julia.quantecon.org/intro.html
- Macro-oriented reference: [Caraiani(2018)].
- Early comparison of languages used in macro (including Julia): [Aruoba and Fernández-Villaverde(2015)].
- FRB NY DSGE model is coded in Julia: https://frbny-dsge.github.io/DSGE.jl/latest/
- Dynare Julia reference manual: https://dynarejulia.github.io/Dynare.jl/dev/

Julia at a Glance

- High-level syntax with near C/Fortran speed (JIT + multiple dispatch).
- Designed for scientific computing; open-source MIT license.
- Interops with C/Fortran/Python; strong linear algebra foundations.

Installing Julia

1. Download from julialang.org/downloads.
2. Install to a path without spaces (Windows suggestion: `C:\Julia\Julia-1.x.x`).
3. Launch Julia: REPL appears; test with `?` for help.

The REPL: Read–Eval–Print–Loop

- Modes: julia> for code, press `?` for help, press `;` for shell, press `]` for package manager (prompt shows as pkg>).
- Use as a calculator; explore docs: `?sum`, `?plot`.

```
help?> sum
julia> x = 1 + 2
3
```

Packages with Pkg

```
julia> import Pkg
julia> Pkg.add("DataFrames")
julia> Pkg.add(["CSV","Plots"])  # multiple at once
julia> using DataFrames, CSV, Plots
```

- `Pkg.status()` to audit environments; use `Project.toml` for reproducibility.

Variables and Types

```
x = 1               # Int64
y = 1.0             # Float64
z = 2 + 3im         # Complex{Int64}
typeof(z)
```

- Dynamic but typed: types are inferred; can annotate when needed.
- Integers, floats, bools, complex, strings.

Operators and Math

```
1 + 5
2 * 3
4 / 2
sqrt(9); exp(1); log(10)
round(1.2)
```

Vectors

```
v = Float64[]          # empty
v1 = zeros(5)
v2 = ones(2)
v3 = collect(range(1, stop=2, length=5))
append!(v1, v2)
```

Matrices and Arrays

```
A = [1 2 3; 4 5 6]
size(A), ndims(A), eltype(A)
R = reshape(range(0, 1, length=4), 2, 2)
```

- Indexing starts at 1; `size`, `ndims`, `eltype`.
- Concatenate with `hcat`, `vcat`.

Indexing and Slicing

```
A = [1 2 3; 4 5 6]
A[1, :]   # first row
A[:, 2]   # second column
A[[1,2], [1,3]]
```

Functions

```
function f(x, y, z)
x + y + z
end
f2(x, y, z) = x + y + z
weighted(x, y; a=1, b=2) = a*x + b*y
```

Anonymous Functions and Map

```
map(x -> x + 1, [1,2,3])
```

```
map([1,2,3]) do x
x + 1
end
```

Returning Multiple Values

```
function stats(a, b, c)
s = a + b + c
p = a * b * c
return s, p
end
s, p = stats(1,2,3)
```

Control Flow: Conditionals

```
function ineq(x, y)
if x > y
2
elseif x < y
1
else
0
end
end
```

Control Flow: Loops

```
j = 3
while j > 0
println(j^2)
j -= 1
end

for k in 1:3
println(k^2)
end
```

Short-Circuit Evaluation

- 1st: uses short-circuit AND
- 2nd: short-circuit OR

```
(x == 0) && error("x must be nonzero")
(x <= 5) || return 0
```

Exceptions

```
f(y) = (y > 0) ? y + 1 : throw(DomainError())
try
f(-2)
catch e
@warn "caught" e
end
```

Tasks (Coroutines) — Idea

- Suspend/resume computations; communicate via `Channel`.
- Useful for producer–consumer patterns.

```
function produce!(c::Channel)
put!(c, "begin")
for x in 1:2
put!(c, x+1)
end
put!(c, "end")
end
chnl = Channel(produce!)
```

Random Numbers

```julia
using Random
Random.seed!(1234)
rand()           # U(0,1)
rand(5)          # vector
randn(2,2)       # N(0,1) matrix
```

Working with Files

```
open("example.txt", "w") do io
write(io, "hello
world
")
end
lines = readlines("example.txt")
```

Paths and Directories

```
homedir()
pwd()
readdir()
cd("..")
```

Delimited Data

```
using CSV, DataFrames
CSV.write("data.csv", DataFrame(x=1:3, y=rand(3)))
df = CSV.read("data.csv", DataFrame)
```

Dates and Times

```
using Dates
Date(2015,5,1)
DateTime(2016,3,10,11)
d = Date("2016-05-05", DateFormat("y-m-d"))
year(d), month(d)
```

DataFrames Basics

```
using DataFrames
DF = DataFrame(x=range(0, 1, length=10), y=randn(10))
first(DF, 5)
describe(DF)
```

Plotting Options

- **Plots.jl** (meta-backend): quick API across backends.
- **PyPlot.jl**: Matplotlib backend.
- **Gadfly.jl**: Grammar-of-graphics style.

```
using Plots
x = range(0, stop=1, length=100); y = randn(100)
plot(x, y, title="Basic Plot")
```

Advanced: Type System

```
struct Point{T}
x::T; y::T
end
norm(p::Point{T}) where {T<:Real} = sqrt(p.x^2 + p.y^2)
```

- Parametric types enable specialization and performance.
- Here, we can compute Euclidean length

Advanced: Multiple Dispatch

```
area(r::Real) = pi*r^2
area(w::Real, h::Real) = w*h
```

- Method chosen by the combination of argument types.

Advanced: Vectorization and Broadcasting

```
x = 1:5; y = 2:6
x .+ y
f(t) = t^2
f.(x)
```

- Vectorization: writing operations so they act on whole arrays at once instead of looping element-by-element explicitly
- Broadcasting: Julia's . syntax that applies a function or operator elementwise across arrays of compatible shapes

Control Flow: Boolean Logic

```
true && false    # false
true || false    # true
!true            # false
(0 < x <= 1)     # chaining comparisons
```

Control Flow: Ternary and If-else Expressions

```
val = x > 0 ? "positive" : "nonpositive"
ans = if x < -1
:low
elseif x <= 1
:mid
else
:high
end
```

Control Flow: Comprehensions

```
# vector and matrix comprehensions
v = [i^2 for i in 1:5]
M = [i*j for i in 1:3, j in 1:3]
# generators (lazy)
sum(i^2 for i in 1:1000)
```

Control Flow: Try/Catch/Finally and @assert

```
try
sqrt(-1)
catch e
@warn "domain error" e
finally
println("cleanup")
end
@assert 2+2 == 4
```

Functions: Methods and Type Annotations

```
area(r::Real) = pi*r^2
area(w::Real, h::Real) = w*h
# dispatch picks method by argument types
area(2.0); area(2,3)
```

Functions: Keyword Args and Defaults

```
function simulate(T; shock=:none, seed=42)
Random.seed!(seed)
# ...
end
simulate(100); simulate(200, shock=:tfp)
```

Functions: Varargs and Splatting

```
sumall(xs...) = foldl(+, xs; init=0)
sumall(1,2,3)
v = (1,2,3)
sumall(v...)   # splat tuple
```

Functions: Mutating Conventions

```
push!(v, x)    # mutates v
sort!(v)       # by convention, ! means mutation
# write your own mutating function
function scale!(v, a)
for i in eachindex(v)
v[i] *= a
end
return v
end
```

Functions: Docstrings and Testing

```julia
"""
f(x,y)
Add two numbers.
"""
f(x,y) = x + y
# rudimentary test
@assert f(2,3) == 5
```

Wrap-Up and Next Steps

- You can navigate Julia, manage packages, and write idiomatic code.
- Coming up: numerical methods and DSGE solution/simulation in Julia.
- Prep: ensure VS Code + Julia extension are installed; test plotting.

S. Borağan Aruoba and Jesús Fernández-Villaverde.
A comparison of programming languages in macroeconomics.
Journal of Economic Dynamics and Control, 58:265–273, 2015.
ISSN 0165-1889.
doi: https://doi.org/10.1016/j.jedc.2015.05.009.
URL https://www.sciencedirect.com/science/article/pii/S0165188915000883.

Petre Caraiani.
Introduction to Quantiative Macroeconomics with Julia.
Academic Press - Elsevier, 1 edition, 2018.

John Stachurski and Thomas Sargent.
Quantitative Macroeconomics with Julia.
Mimeo, 2017.