Numerical Methods in Julia

Course 2 — Short Macroeconomics Course Using Julia

November 26, 2025

Outline

Aims

- Workhorse numerical tools for macro: linear algebra, root finding, optimization, integration, interpolation, ODEs, and Monte Carlo.
- Julia idioms: broadcasting, in-place updates, types, and performance.
- Reproducibility: environments, RNG, and benchmarking.

Guiding Principles

- Stability and conditioning.
- Prefer library functions (e.g. factorizations) to manually built functions.
- Execution time: use `BenchmarkTools.jl` .

Matrix operations: loops vs. broadcast (elementwise)

Goal: add a scaled matrix into another, $C \leftarrow a \cdot A + C$ (AXPY on matrices).

```
using LinearAlgebra
# Broadcasting (concise)
C .+= a .* A
# Full broadcast (no temporaries): equivalent but preferred
@. C = a*A + C           # or: C .+= a .* A
# Loop
function axpy_loop!(C, a, A)
@inbounds @simd for j in axes(C,2), i in axes(C,1)
C[i,j] += a * A[i,j]
end
return C
end
```

Notes. Broadcasting with dotted operators fuses elementwise ops and avoids temporaries. Loops give maximal control (`@inbounds` to skip bounds checks, `@simd` to vectorize).

BLAS and factorizations (dense linear algebra)

Goal A: matrix–matrix and matrix–vector products (use BLAS).

```
# High-level BLAS-backed ops
Y = A * x                   # gemv (matrix-vector)
C = A * B                   # gemm (matrix-matrix)
# In-place BLAS (avoid allocations)
mul!(Y, A, x)               # Y := A*x
mul!(C, A, B, 1.0, 0.0)     # C := 1*A*B + 0*C
```

Goal B: solve many linear systems with the same *A without* re-factorizing.

```
# One-off solve (preferred over inv(A)*B)
X = A \
# Many solves with same A: factor once, reuse
F = lu(A)                   # or cholesky(A) if SPD
X  = F \ B                  # multiple RHS columns
```

Notes. Never form `inv(A)` for solving; use \ or a reusable factorization (`lu`, `cholesky`, `qr`) for speed and numerical stability.

Broadcasting and In-place

```
x = rand(1_000_000)
y = similar(x)
y .= @. 2x + 1  # fused broadcast
```

Avoid Allocations

```
function scale!(y, a, x)
@inbounds @simd for i in eachindex(x)
y[i] = a*x[i]
end
return y
end
```

Why the extras help

- `scale!` — in-place update; no new array is created.
- `eachindex(x,y)` — fast, allocation-free indices that match shapes.
- `@inbounds` — skips bounds checks (you guarantee indices are valid).
- `@simd` — hints LLVM to auto-vectorize the loop for speed.

BenchmarkTools

```
using BenchmarkTools
f(x) = sum(@. 2x + 1)
x = rand(10_000)
@btime f($x)
```

Why the extras help

- Why the $ in `@btime f($x)`:
- The dollar *uses* `x` into the benchmark.
- That means BenchmarkTools substitutes the current value of `x` directly into the tested expression so the benchmark doesn't include (1) global-variable lookup and (2) spurious allocations from treating `x` as a non-constant global.

Type Stability

```
# inconsistent return types hurt performance
bad(x) = x > 0 ? 1 : 1.0
# fix by consistent types
good(x) = x > 0 ? 1.0 : 1.0
```

Floating Point Essentials

```julia
using Printf
x = 0.1 + 0.2
@printf("%.17f\n", x)  # 0.30000000000000004
isapprox(x, 0.3; atol=1e-12)
```

- Use `isapprox` with tolerances.

Conditioning and Stability

```
A = [1.0 1.0; 1.0 1.000001]
condA = cond(A)   # 2-norm condition number
```

- Bad conditioning magnifies errors.
- Prefer stable algorithms (QR, SVD) over naive ones (which work, but are numerically unstable).

BLAS-backed Operations

```
using LinearAlgebra
A = randn(3,3); b = randn(3)
A*b           # matrix-vector
A*A'          # matrix-matrix
```

Solve, Do Not Invert

```
A = randn(3,3); b = randn(3)
x = A \ b             # solve Ax = b
# If repeated solves with same A, factorize once:
F = lu(A); x2 = F \ b
```

Factorizations

```
F = qr(randn(4,3))
Q = Matrix(F.Q); R = F.R
S = svd(randn(4,3))
U, s, Vt = S.U, S.S, S.Vt
```

Eigenvalues and Symmetry

```
A = Symmetric(randn(4,4)); A = A*A'  # SPD
D, V = eigen(A)  # eigen decomposition
```

- symmetric is a function from LinearAlgebra telling Julia to treat an object as symmetric.
- enables safer and faster algorithms.

Sparse Matrices

```
using SparseArrays
S = spdiagm(0 => fill(2.0,5), 1 => fill(-1.0,4), -1 => fill(-1.0,4))
b = ones(5)
x = S \ b
```

- sparse matrices: arrays where most entries are zero.
- A sparse matrix in Julia is usually CSC (Compressed Sparse Column) format.

Fixed Point Iteration

```
# Solve x = cos(x)
x = 0.5
for k in 1:30
x = cos(x)
end
x
```

- $f(x) = 0$ is mathematically equivalent to looking for a fixed point for which $x = g(x)$ when we rewrite the problem as:
- $x = x - f(x)$.
- To apply the function iteration, we need a starting point which must be reasonably close to the root of f (x).

Newton Method (Scalar)

```
# Solve f(x) = x^2 - 2 = 0
f(x) = x^2 - 2
fp(x) = 2x
x = 1.0
for k in 1:10
x -= f(x)/fp(x)
end
```

- The algorithm of the Newton's method is pretty straightforward as the linearization is done using the Taylor expansion.
- Assume a function f and an initial guess of the root given by x0. We create an iteration equation on x_{k+1} given $x + k$ using a Taylor approximation of order one to replace f (x):
- $f(x) \approx f(x_k) + f'(x_k)(x - x_k) = 0$ which gives the update formula:
- $x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$.

Newton Method (Vector)

```
using LinearAlgebra
function newton(F, J, x; iters=20)
for k in 1:iters
x -= J(x) \ F(x)
end
return x
end
```

Unconstrained Optimization

```
# Gradient descent demo
f(x) = (x-2)^2
fp(x) = 2(x-2)
x = 0.0
for k in 1:20
x -= 0.2*fp(x)
end
x
```

Line Search Template

```
function linesearch(f, x; d=-1.0, alpha=1.0, beta=0.5)
fx = f(x)
while f(x + alpha*d) > fx
alpha *= beta
end
return alpha
end
```

- A simple backtracking line search: start with step α
- repeatedly shrink it by β until moving from x to $x + \alpha$ d actually lowers f, then return that step size.

Quasi-Newton Sketch

```
# Illustrative only
function steepest_descent(f, g, x; steps=30)
for k in 1:steps
d = -g(x)
a = 0.1
x += a*d
end
return x
end
```

- Moves iteratively in the negative gradient direction $d = -g(x)$ — the direction of steepest local decrease
- $a = 0.1$ each time, for steps iterations, to minimize f

Finite Differences

```
# Forward difference
fd(f, x, h=1e-6) = (f(x+h) - f(x)) / h
fd(sin, 0.3)
```

Central Differences

```
cd(f, x, h=1e-6) = (f(x+h) - f(x-h)) / (2h)
cd(exp, 0.0)
```

Composite Simpson Rule

```
function simpson(f, a, b, n)
n % 2 == 0 || error("n must be even")
h = (b-a)/n
s = f(a) + f(b)
for k in 1:2:n-1
s += 4*f(a + k*h)
end
for k in 2:2:n-2
s += 2*f(a + k*h)
end
return s*h/3
end
simpson(x->exp(-x^2), 0.0, 1.0, 200)
```

Piecewise Linear Interpolation

```
x = 0:0.5:5
y = @. sin(x) + 0.1*x
function lininterp(xg, yg, x)
i = searchsortedlast(xg, x)
i == length(xg) && (i -= 1)
t = (x - xg[i])/(xg[i+1]-xg[i])
return (1-t)*yg[i] + t*yg[i+1]
end
lininterp(collect(x), collect(y), 2.2)
```

- piecewise-linear interpolator
- given a sorted grid xg with values yg, it returns an approximate y at any x by joining neighboring points with straight lines.

Using Package Interpolations

```
#use Interpolations
using Interpolations

#create some data
A_x = 1.0:4.0:50.0

A   = collect(2 .* A_x)

itp  = interpolate(A, BSpline(Cubic(Line())), OnGrid())
sitp = scale(itp, A_x)

sitp(3.0)
```

- implements a variety of interpolation algorithms.
- supports B-splines and irregular grids.

Explicit Euler

```
# x' = -x, x(0)=1
f(t,x) = -x
function euler(f, t0, x0, h, n)
t = t0; x = x0
for k in 1:n
x += h*f(t, x)
t += h
end
return x
end
euler(f, 0.0, 1.0, 0.01, 100)
```

Explicit Euler

- **Forward Euler for** $x'(t) = f(t,x)$: starting at (t_0, x_0), iterate

$$x_{k+1} = x_k + h\,f(t_k, x_k), \qquad t_{k+1} = t_k + h,$$

for $k = 0, \dots, n-1$.

- For $f(t,x) = -x$, $x(0) = 1$, $h = 0.01$, $n = 100$, the method approximates

$$x(1) \approx (1 - 0.01)^{100} \approx 0.366,$$

close to the exact value $e^{-1} \approx 0.368$.

Runge–Kutta 4

```
function rk4(f, t0, x0, h, n)
t = t0; x = x0
for k in 1:n
k1 = f(t, x)
k2 = f(t + h/2, x + h*k1/2)
k3 = f(t + h/2, x + h*k2/2)
k4 = f(t + h, x + h*k3)
x += h*(k1 + 2k2 + 2k3 + k4)/6
t += h
end
return x
end
rk4(f, 0.0, 1.0, 0.01, 100)
```

Runge–Kutta 4 Algorithm

- **RK4 update for** $x'(t) = f(t,x)$: from (t_n, x_n) with step h,

$$\begin{aligned} k_1 &= f(t_n,\ x_n), \\ k_2 &= f\left(t_n + \tfrac{h}{2},\ x_n + \tfrac{h}{2}k_1\right), \\ k_3 &= f\left(t_n + \tfrac{h}{2},\ x_n + \tfrac{h}{2}k_2\right), \\ k_4 &= f(t_n + h,\ x_n + h\,k_3), \end{aligned}$$

$$x_{n+1} = x_n + \tfrac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4), \qquad t_{n+1} = t_n + h.$$

  It samples the slope at four locations within the step and takes a weighted average, greatly improving accuracy over forward Euler.

- **Accuracy and use:** RK4 has a good accuracy–cost tradeoff for smooth f. It is explicit, easy to implement, and typically stable for moderate h.

RNG Control

```
using Random
Random.seed!(2025)
rand(), randn()
```

Monte Carlo Integration

```
# Estimate E[g(Z)] for Z ~ N(0,1)
using Statistics
function mc(g, N=10_000)
z = randn(N)
return mean(g.(z))
end
mc(z->z^2)
```

- **Monte Carlo integration:** The function `mc(g,N)` draws N iid samples $Z_1, \ldots, Z_N \sim \mathcal{N}(0,1)$ and returns the sample mean $\frac{1}{N}\sum_{i=1}^{N} g(Z_i)$. By the Law of Large Numbers, this converges to $\mathbb{E}[g(Z)]$ as $N \to \infty$. Broadcasting `g.(z)` applies g elementwise to the vector of draws.
- **Example:** `mc(z->z^2)` estimates $\mathbb{E}[Z^2]$ for $Z \sim \mathcal{N}(0,1)$, which equals 1 (the variance). With large N, the return should be close to 1; the error decays on the order of $1/\sqrt{N}$.

Variance Reduction: Antithetic

```
function mc_anti(g, N=10_000)
z = randn(div(N,2))
return mean((g.(z) .+ g.(-z))/2)
end
```

- **Construction.** For $Z \sim \mathcal{N}(0,1)$, draw z_i and pair it with its mirror $-z_i$; estimate $\mathbb{E}[g(Z)]$ by $\hat{\mu} = \dfrac{1}{N}\displaystyle\sum_{i=1}^{N/2}\left[g(z_i) + g(-z_i)\right]$ (unbiased since $Z \overset{d}{=} -Z$).
- **Why it helps.** The pair induces negative correlation when g is roughly monotone, giving $\mathrm{Var}\left(\frac{g(Z)+g(-Z)}{2}\right) = \frac{\mathrm{Var}(g(Z))}{2}(1+\rho)$ with $\rho = \mathrm{corr}(g(Z), g(-Z)) < 0$.

Summary

- Covered numerical methods.
- Emphasized stability, conditioning, and reproducibility.
- Next chapter: solving and simulating DSGE models.