Numerical Linear Algebra with Iterative Methods (Julia)

Machine Learning Fundamentals for Economists

Jesse Perla

jesse.perla@ubc.ca

University of British Columbia

Table of contents

Overview

Motivation

- In preparation for the ML lectures we cover some core numerical linear algebra concepts
- Many of these are directly useful
  - → e.g. solving large LLS and systems of equations, such as you might find with a large scale two-way fixed effects model
  - → Solving systems of equations is useful in itself
- Others will be helpful in setting up understanding for ML
  - → Matrix-free and iterative methods
  - → What governs complexity and convergence speed
  - → Conditioning
  - → Regularization

Summary and Material

- See QuantEcon Krylov Methods and Matrix Conditioning

```
using LinearAlgebra, Statistics, BenchmarkTools, SparseArrays, Random
using LaTeXStrings, Plots, IterativeSolvers, Preconditioners, IncompleteLU, LinearMaps
using Arpack
Random.seed!(42);  # seed random numbers for reproducibility
```

```
Precompiling packages...
   2636.2 ms  ✓ IterativeSolvers
  1 dependency successfully precompiled in 3 seconds. 10 already precompiled.
Precompiling packages...
    548.2 ms  ✓ CommonSolve
   1098.4 ms  ✓ AMD
    984.8 ms  ✓ LimitedLDLFactorizations
   2438.6 ms  ✓ AlgebraicMultigrid
   1735.3 ms  ✓ Preconditioners
  5 dependencies successfully precompiled in 5 seconds. 6 already precompiled.
Precompiling packages...
    633.7 ms  ✓ IncompleteLU
  1 dependency successfully precompiled in 1 seconds. 3 already precompiled.
Precompiling packages...
   1579.9 ms  ✓ LinearMaps
  1 dependency successfully precompiled in 2 seconds
Precompiling packages...
    411.8 ms  ✓ LinearMaps → LinearMapsStatisticsExt
  1 dependency successfully precompiled in 1 seconds. 2 already precompiled.
Precompiling packages...
    986.5 ms  ✓ LinearMaps → LinearMapsSparseArraysExt
  1 dependency successfully precompiled in 1 seconds. 4 already precompiled.
```

Conditioning

Direct Methods and Conditioning

- Some algorithms and some matrices are more numerically stable than others
  - → By “numerically stable” we mean sensitive to accumulated roundoff errors
- A key issue is when matrices are close to singular, or almost have collinear columns. Many times this can’t be avoided, other times it can (e.g., choose orthogonal polynomials rather than monomials)
- This will become even more of an issue with iterative methods, but is also the key to rapid convergence. Hint: $Ax = b$ is easy if $A = I$, even if it is dense.

Condition Numbers of Matrices

- $\det(A) \approx 0$ may say it is "almost" singular, but it is not scale-invariant
- The condition number κ, given matrix norm $\|\cdot\|$ uses the matrix norm

$$\text{cond}(A) \equiv \|A\|\|A^{-1}\| \geq 1$$

- Expensive to calculate, can show that given spectrum

$$\text{cond}(A) = \left|\frac{\lambda_{max}}{\lambda_{min}}\right|$$

- Intuition: if $\text{cond}(A) = K$, then $b \to b + \nabla b$ change in b amplifies to a $x \to x + K\nabla b$ error when solving $Ax = b$.
- See Matlab Docs on inv for why `inv` is a bad idea when $\text{cond}(A)$ is huge

Condition Numbers and Matrix Operations

- The identity matrix is as good as it gets
- Otherwise, the issue is when matrices are of fundamentally different scales

```
@show cond(I(2))
epsilon = 1E-6
A2 = [1.0 0.0
     1.0 epsilon]
@show cond(A2);
@show cond(A2');
@show cond(inv(A2));
```

```
cond(I(2)) = 1.0
cond(A2) = 2.0000000000005004e6
cond(A2') = 2.0000000000004997e6
cond(inv(A2)) = 2.0000000023323308e6
```

Conditioning Under Matrix Products

- Matrix operations can often amplify the condition number, or may be invariant
- Be especially careful with normal equations/etc.

```
1 lauchli(N, epsilon) = [ones(N)';
2                        epsilon * I(N)]'
3 epsilon = 1E-8
4 L = lauchli(3, epsilon) |> Matrix
5 @show cond(L)
6 @show cond(L' * L)
7 L
```

```
cond(L) = 1.732050807568878e8
cond(L' * L) = 2.8104131146758097e32
3×4 Matrix{Float64}:
 1.0  1.0e-8  0.0     0.0
 1.0  0.0     1.0e-8  0.0
 1.0  0.0     0.0     1.0e-8
```

See here for why a monomial basis is a bad idea

Stationary Iterative Methods

Direct Methods

- Direct methods work with a matrix, stored in memory, and typically involve factorizations
  - → Can be dense or sparse
  - → They can be fast, and solve problems to machine precision
- Typically are superior until problems get large or have particular structure
- But always use the right factorizations and matrix structure! (e.g., posdef, sparse, etc)
- The key limitations are the sizes of the matrices (or the sparsity)

Iterative Methods

- Iterative methods are in the spirit of gradient descent and optimization algorithms
  - → They take an initial guess and update until convergence
  - → They work on matrix-vector and vector-matrix products, and can be **matrix-free**, which is a huge advantage for huge problems
  - → Rather than waiting until completion like direct methods, you can control stopping
- The key limitations on performance are geometric (e.g., conditioning), not dimensionality
- Two rough types: stationary methods and Krylov methods

Example from Previous Lectures

- Variation on CTMC example: $a > 0$ gain, $b > 0$ to lose
- Solve the Bellman Equation for a CTMC

```
N = 100
a = 0.1
b = 0.05
rho = 0.05
Q = Tridiagonal(fill(b, N-1),
                [-a; fill(-(a + b), N-2); -b],
                fill(a, N-1))

r = range(0.0, 10.0, length = N)
A = rho * I - Q
v_direct = A \ r
mean(v_direct)
```

```
101.96306207828795
```

Diagonal Dominance

- Stationary Iterative Methods reorganize the problem so it is a contraction mapping and then iterate
- For matrices that are **strictly diagonal dominant**

$$|A_{ii}| \geq \sum_{j \neq i} |A_{ij}| \quad \text{for all } i = 1 \ldots N$$

→ i.e., sum of all off-diagonal elements in a row is less than the diagonal element in absolute value

- Note for our problem rows sum to 0 so if $\rho > 0$ then $\rho I - Q$ is strictly diagonally dominant

Jacobi Iteration

- To solve a system $Ax = b$, split the matrix A into its diagonal and off-diagonal elements. That is,

$$A \equiv D + R$$

$$D \equiv \begin{bmatrix} A_{11} & 0 & \dots & 0 \\ 0 & A_{22} & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & A_{NN} \end{bmatrix} \quad R \equiv \begin{bmatrix} 0 & A_{12} & \dots & A_{1N} \\ A_{21} & 0 & \dots & A_{2N} \\ \vdots & \vdots & \vdots & \vdots \\ A_{N1} & A_{N2} & \dots & 0 \end{bmatrix}$$

Jacobi Iteration Algorithm

- Then we can rewrite $(D + R)x = b$ as

$$Dx = b - Rx$$
$$x = D^{-1}(b - Rx)$$

Where D^{-1} is trivial since diagonal. To solve, take an iteration x^k, starting from x^0,

$$x^{k+1} = D^{-1}(b - Rx^k)$$

Code for Jacobi Iteration

- Showing Jacobi Iteration and a better method, successive over-relaxation (SOR). Many better algoriths exist

```
v = zeros(N)

jacobi!(v, A, r, maxiter = 40)
@show norm(v - v_direct, Inf)
sor!(v, A, r, 1.1, maxiter = 40)
@show norm(v - v_direct, Inf);
```

```
norm(v - v_direct, Inf) = 0.0017762754968373429
norm(v - v_direct, Inf) = 9.052314453583676e-12
```

Krylov Methods

Krylov Subspaces

- Krylov methods are a class of iterative methods that use a sequence of subspaces
- The subspaces are generated by repeated matrix-vector products
  - → i.e., given an A and a initial value b we could generate the sequence
  - → $b, Ab, A^2b, \dots, A^kb$ and see
- Note that the only operation we require from our linear operator A is the matrix-vector product. This is a huge advantage for large problems
- e.g. Krylov method is **Conjugate Gradient** for posdef A

Conjugate Gradient

- Solving this system with the conjugate gradient method
- Using matrix, but could just implement A as a function

```
1 N = 100
2 A = sprand(100, 100, 0.1)
3 A = A * A'  # easy posdef
4 b = rand(N)
5 x_direct = A \ b
6 @show cond(Matrix(A * A'))
7 x = zeros(N)
8 sol = cg!(x, A, b, log = true, maxiter = 1000)
9 sol[end]
```

```
cond(Matrix(A * A')) = 6.933646354576138e17
Not converged after 1000 iterations.
```

Iterative Methods for LLS

- LSMR is one of several Krylov methods for solving LLS

$$\min_{\beta} ||X\beta - y||^2 + \alpha||\beta||^2$$

- Where $\alpha \geq 0$. If $\alpha = 0$ then it is delivers the ridgeless regression limit, even if underdetermined

LSMR Example

```
M = 1000
N = 10000
sigma = 0.1
beta = rand(M)
# simulate data
X = sprand(N, M, 0.1)
y = X * beta + sigma * randn(N)
beta_direct = X \ y
results = lsmr(X, y, log = true)
beta_lsmr = results[1]
@show norm(beta_direct - beta_lsmr)
println("$(results[end])")
```

```
norm(beta_direct - beta_lsmr) = 1.0103408119554285e-5
Converged after 14 iterations.
```

Matrix-Free LLS

- To solve LLS problems, we need Xu and $X^T v$ products
- We can provide those functions directly (cheating here by just using the matrix itself)

```julia
# matrix-vector product
X_func(u) = X * u

# adjoint-vector product
X_T_func(v) = X' * v

X_map = LinearMap(X_func, X_T_func, N, M)
results = lsmr(X_map, y, log = true)
println("$(results[end])")
```

```
Converged after 14 iterations.
```

Eigenvalue Problems

- Variation on CTMC example: $a > 0$ gain, $b > 0$ to lose

```
N = 4
a = 0.1
b = 0.05
Q = Tridiagonal(fill(b, N-1),
                [-a; fill(-(a + b), N-2); -b],
                fill(a, N-1))
# Find smallest magnitude eigenvalue (i.e. 0)
lambda, phi = eigs( Q', nev = 1, which = :SM, maxiter = 1000)
phi = real(phi) ./ sum(real(phi))
@show lambda
@show mean(phi);
Q'
```

```
lambda = ComplexF64[-2.6156219089047246e-17 + 0.0im]
mean(phi) = 0.25
```

```
4×4 Tridiagonal{Float64, Vector{Float64}}:
 -0.1   0.05    ·      ·
  0.1  -0.15   0.05    ·
   ·    0.1   -0.15   0.05
   ·     ·     0.1   -0.05
```

Implementing Matrix-Free Operator for Adjoint

```
function Q_adj_product(x)
    Q_x = zero(x)
    Q_x[1] = -a * x[1] + b * x[2]
    for i = 2:(N-1)
        Q_x[i] = a * x[i-1] - (a + b) * x[i] + b * x[i+1]
    end
    Q_x[N] = a * x[N-1] - b * x[N]
    return Q_x
end
x_check = rand(N)
norm(Q_adj_product(x_check) - Q' * x_check)
```

```
0.0
```

Solving with a Wrapper for the Matrix-Free Operator

- The `LinearMap` wrapper adds features required for algorithms (e.g. `size(Q_adj_map` and `Q_adj_map * v` overloads)

```
1 Q_adj_map = LinearMap(Q_adj_product, N)
2 # Get smallest magnitude only using the Q'(x) map
3 lambda, phi = eigs(Q_adj_map, nev = 1, which = :SM, maxiter = 1000)
4 phi = real(phi) ./ sum(real(phi))
5 @show lambda
6 @show mean(phi);
```

```
lambda = ComplexF64[1.5057759750243998e-17 + 0.0im]
mean(phi) = 0.25
```

Preconditioning

Changing the Geometry

- In practice, most Krylov methods are preconditioned in practice or else direct methods usually dominate. Same with large nonlinear systems
- As discussed, the key issue for the convergence speed of iterative methods is the geometry (e.g. condition number of hessian, etc)
- Preconditioning changes the geometry. e.g. more like circles or with eigenvalue problems spread out the eigenvalues of interest
- Preconditioners for a matrix A requires art and tradeoffs
  - → Want be relatively cheap to calculate, and must be invertible
  - → Want to have $\mathbf{cond}(PA) \ll \mathbf{cond}(A)$
- Ideal preconditioner for $Ax = b$ is $P = A^{-1}$ since $A^{-1}Ax = x = A^{-1}b$
  - → $\mathbf{cond}(A^{-1}A) = \mathbf{1}$! But that is equivalent to solving problem

Right-Preconditioning a Linear System

$$
\begin{aligned}
Ax &= b \\
AP^{-1}Px &= b \\
AP^{-1}y &= b \\
Px &= y
\end{aligned}
$$

That is, solve $(AP^{-1})y = b$ for y, and then solve $Px = y$ for x.

Raw Conjugate Gradient

```
1 N = 200
2 A = sprand(N, N, 0.1)   # 10 percent non-zeros
3 A = A * A'
4 b = rand(N)
5 @show cond(Matrix(A))
6 sol = cg(A, b, log = true, maxiter = 1000)
7 sol[end]
```

```
cond(Matrix(A)) = 724437.8616697111
Converged after 386 iterations.
```

Diagonal Preconditioner

- A simple preconditioner is the diagonal of A
- This is cheap to calculate, and is invertible if the diagonal has no zeros
- For some problems this has a huge impact on convergence/condition, for others it does nothing

```
P = DiagonalPreconditioner(A)
sol = cg(A, b; Pl = P, log = true, maxiter = 1000)
sol[end]
```

```
Converged after 363 iterations.
```

Incomplete LU or Cholesky

- Iterate part of the way on an LU or Cholesky factorization
- Not the total inverse, but can make a big difference

```
P = ilu(A, τ = 0.1)
sol = cg(A, b; Pl = P, log = true, maxiter = 1000)
sol[end]
```

```
Converged after 154 iterations.
```

Others

- In the above we aren't getting huge gains, but it is also lacking structure
- If you have problems with multiple scales, as might come out of discretizing multiple dimensions in a statepsace, see **multigrid** methods
  - → Algebraic Multigrid preconditioner is often useful even outside of having different scales
- Other preconditioners include ones intended for **Graph Laplacians** such as approximate cholesky decompositions and combinatorial multigrid preconditioners.
  - → See **paper** for more