Direct Methods and Matrix Factorizations (Julia)

Machine Learning Fundamentals for Economists

Jesse Perla

jesse.perla@ubc.ca

University of British Columbia

Table of contents

- Overview
- Complexity
- Matrix Structure
- Factorizations
- Continuous Time Markov Chains

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 systems of equations or as building blocks in bigger algorithms

Packages and Materials

- See QuantEcon Numerical Linear Algebra and associated notebooks

```
using LinearAlgebra, Statistics, BenchmarkTools, SparseArrays, Random
using Plots
Random.seed!(42);  # seed random numbers for reproducibility
```

Complexity

Basic Computational Complexity

Big-O Notation

For a function $f(N)$ and a positive constant C, we say $f(N)$ is $O(g(N))$, if there exist positive constants C and N_0 such that:

$$0 \leq f(N) \leq C \cdot g(N) \quad \text{for all } N \geq N_0$$

- Often crucial to know how problems scale asymptotically (as $N \to \infty$)
- Caution! This is only an asymptotic limit, and can be misleading for small N
  - → $f_1(N) = N^3 + N$ is $O(N^3)$
  - → $f_2(N) = 1000N^2 + 3N$ is $O(N^2)$
  - → For roughly $N > 1000$ use f_2 algorithm, otherwise f_1

Examples of Computational Complexity

- Simple examples:
  - → $x \cdot y = \sum_{n=1}^{N} x_n y_n$ is $O(N)$ since it requires N multiplications and additions
  - → Ax for $A \in \mathbb{R}^{N \times N}, x \in \mathbb{R}^N$ is $O(N^2)$ since it requires N dot products, each $O(N)$

Computational Complexity

Ask yourself whether the following is a **computationally expensive** operation as the matrix **size increases**

- Multiplying two matrices?
  - → *Answer*: It depends. Multiplying two diagonal matrices is trivial.
- Solving a linear system of equations?
  - → *Answer*: It depends. If the matrix is the identity, the solution is the vector itself.
- Finding the eigenvalues of a matrix?
  - → *Answer*: It depends. The eigenvalues of a triangular matrix are the diagonal elements.

Numerical Precision

Machine Epsilon

For a given datatype, ϵ is defined as $\epsilon = \min_{\delta>0} \{\delta : 1 + \delta > 1\}$

- Computers have finite precision. 64-bit typical, but 32-bit on GPUs

```
machine epsilon for float64 = 2.220446049250313e-16
1 + eps/2 == 1? true
machine epsilon for float32 = 1.1920929e-7
```

Matrix Structure

Matrix Structure

- A key principle is to ensure you don't lose "structure"
  - → e.g. if sparse, operations should keep it sparse if possible
  - → If triangular, then use appropriate algorithms instead of converting back to a dense matrix
- Key structure is:
  - → Symmetry, diagonal, tridiagonal, banded, sparse, positive-definite
- The worse operations for losing structure are matrix multiplication and inversion

Example Losing Sparsity

- Here the density increases substantially

```
A = sprand(10, 10, 0.45)  # random sparse 10x10, 45 percent filled with non-zeros

@show nnz(A)  # counts the number of non-zeros
invA = sparse(inv(Array(A)))  # Julia won't invert sparse, so convert to dense with Array.
@show nnz(invA);
```

```
nnz(A) = 46
nnz(invA) = 100
```

Losing Tridiagonal Structure

- An even more extreme example. Tridiagonal has roughly $3N$ nonzeros. Inverses are dense N^2

```
1 N = 5
2 A = Tridiagonal([fill(0.1, N - 2); 0.2], fill(0.8, N), [0.2; fill(0.1, N - 2)])
3 inv(A)
```

```
5×5 Matrix{Float64}:
  1.29099      -0.327957     0.0416667  -0.00537634   0.000672043
 -0.163978      1.31183     -0.166667    0.0215054   -0.00268817
  0.0208333    -0.166667     1.29167    -0.166667     0.0208333
 -0.00268817    0.0215054   -0.166667    1.31183     -0.163978
  0.000672043  -0.00537634   0.0416667  -0.327957     1.29099
```

Forming the Covariance and/or Gram Matrix

- Another common example is $A'A$

```
1 A = sprand(20, 21, 0.3)
2 @show nnz(A) / 20^2
3 @show nnz(A' * A) / 21^2;
```

```
nnz(A) / 20 ^ 2 = 0.34
nnz(A' * A) / 21 ^ 2 = 0.9229024943310657
```

Specialized Algorithms

Besides sparsity/storage, the real loss is you miss out on algorithms. For example, lets setup the benchmarking code

```
using BenchmarkTools
function benchmark_solve(A, b)
    println("A\\b for typeof(A) = $(string(typeof(A)))")
    @btime $A \ $b
end
```

```
benchmark_solve (generic function with 1 method)
```

Compare Dense vs. Sparse vs. Tridiagonal

```
N = 1000
b = rand(N)
A = Tridiagonal([fill(0.1, N - 2); 0.2], fill(0.8, N), [0.2; fill(0.1, N - 2)])
A_sparse = sparse(A)  # sparse but losing tridiagonal structure
A_dense = Array(A)    # dropping the sparsity structure, dense 1000x1000

# benchmark solution to system A x = b
benchmark_solve(A, b)
benchmark_solve(A_sparse, b)
benchmark_solve(A_dense, b);
```

```
A\b for typeof(A) = LinearAlgebra.Tridiagonal{Float64, Vector{Float64}}
  26.860 μs (20 allocations: 47.39 KiB)
A\b for typeof(A) = SparseArrays.SparseMatrixCSC{Float64, Int64}
  512.290 μs (95 allocations: 1.03 MiB)
A\b for typeof(A) = Matrix{Float64}
  19.588 ms (9 allocations: 7.64 MiB)
```

Triangular Matrices and Back/Forward Substitution

- A key example of a better algorithm is for triangular matrices
- Upper or lower triangular matrices can be solved in $O(N^2)$ instead of $O(N^3)$

```
b = [1.0, 2.0, 3.0]
U = UpperTriangular([1.0 2.0 3.0;
                     0.0 5.0 6.0;
                     0.0 0.0 9.0])
U \ b
```

```
3-element Vector{Float64}:
 0.0
 0.0
 0.3333333333333333
```

Backwards Substitution Example

$$Ux = b$$

$$U \equiv \begin{bmatrix} 3 & 1 \\ 0 & 2 \end{bmatrix}, \quad b = \begin{bmatrix} 7 \\ 2 \end{bmatrix}$$

Solving bottom row for x_2

$$2x_2 = 2, \quad x_2 = 1$$

Move up a row, solving for x_1, substituting for x_2

$$3x_1 + 1x_2 = 7, \quad 3x_1 + 1 \times 1 = 7, \quad x_1 = 2$$

Generalizes to many rows. For L it is “forward substitution”

Factorizations

Factorizing matrices

- Just like you can factor $6 = 2 \cdot 3$, you can factor matrices
- Unlike integers, you have more choice over the properties of the factors
- Many operations (e.g., solving systems of equations, finding eigenvalues, inverting, finding determinants) have a factorization done internally
  - → Instead you can often just find the factorization and reuse it
- Key factorizations: LU, QR, Cholesky, SVD, Schur, Eigenvalue

LU(P) Decompositions

- We can "factor" any square A into $PA = LU$ for triangular L and U. Invertible can have $A = LU$, called the LU decomposition. "P" is for partial-pivoting
- Singular matrices may not have full-rank L or U matrices

```
1 N = 4
2 A = rand(N, N)
3 b = rand(N)
4 # chooses the right factorization based on matrix structure
5 # LU here
6 Af = factorize(A)
7 Af.P * A ≈ Af.L * Af.U
```

```
true
```

Using a Factorization

- In Julia the factorization objects typically overload the `\` and functions such as `inv`

```
1 @show Af \ b
2 @show inv(Af) * b
```

```
Af \ b = [1.567631093835083, -1.8670423474177864, -0.7020922312927874, 1.0653095651070625]
inv(Af) * b = [1.5676310938350828, -1.8670423474177873, -0.7020922312927873, 1.0653095651070625]
```

```
4-element Vector{Float64}:
  1.5676310938350828
 -1.8670423474177873
 -0.7020922312927873
  1.0653095651070625
```

LU Decompositions and Systems of Equations

- Pivoting is typically implied when talking about “LU”
- Used in the default `solve` algorithm (without more structure)
- Solving systems of equations with triangular matrices: for $Ax = LUx = b$
  1. Define $y = Ux$
  2. Solve $Ly = b$ for y and $Ux = y$ for x
- Since both are triangular, process is $O(N^2)$ (but LU itself $O(N^3)$)
- Could be used to find `inv`
  - → $A = LU$ then $AA^{-1} = I = LUA^{-1} = I$
  - → Solve for Y in $LY = I$, then solve $UA^{-1} = Y$
- Tight connection to textbook Gaussian elimination (including pivoting)

Cholesky

- LU is for general invertible matrices, but it doesn't use positive-definiteness or symmetry
- The Cholesky is the right factorization for general positive-definite matrices. For general symmetric matrices you can use Bunch-Kaufman or others
- $A = LL'$ for lower triangular L or equivalent for upper triangular

```
N = 500
B = rand(N, N)
A_dense = B' * B  # an easy way to generate a symmetric positive semi-definite matrix
A = Symmetric(A_dense)  # flags the matrix as symmetric
println("A is symmetric? $(issymmetric(A))")
```

```
A is symmetric? true
```

Comparing Cholesky

- By default it doesn't know the matrix is positive-definite, so `factorize` is the best it can do given symmetric structure

```
b = rand(N)
factorize(A) |> typeof
cholesky(A) \ b  # use the factorization to solve

benchmark_solve(A, b)
benchmark_solve(A_dense, b)
@btime cholesky($A, check = false) \ $b;
```

```
A\b for typeof(A) = LinearAlgebra.Symmetric{Float64, Matrix{Float64}}
  2.049 ms (13 allocations: 2.16 MiB)
A\b for typeof(A) = Matrix{Float64}
  2.987 ms (9 allocations: 1.92 MiB)
  1.611 ms (6 allocations: 1.91 MiB)
```

Eigen Decomposition

- For square, symmetric, non-singular matrix A factor into

$$A = Q\Lambda Q^{-1}$$

- Q is a matrix of eigenvectors, Λ is a diagonal matrix of paired eigenvalues
- For symmetric matrices, the eigenvectors are orthogonal and $Q^{-1}Q = Q^TQ = I$ which form an orthonormal basis
- Orthogonal matrices can be thought of as rotations without stretching
- More general matrices all have a Singular Value Decomposition (SVD)
- With symmetric A, an interpretation of Ax is that we can first rotate x into the Q basis, then stretch by Λ, then rotate back

Calculating the Eigen Decomposition

```
1 A = Symmetric(rand(5, 5))  # symmetric matrices have real eigenvectors/eigenvalues
2 A_eig = eigen(A)
3 Λ = Diagonal(A_eig.values)
4 Q = A_eig.vectors
5 @show norm(Q * Λ * inv(Q) - A)
6 @show norm(Q * Λ * Q' - A);
```

```
norm(Q * Λ * inv(Q) - A) = 5.243912693681636e-15
norm(Q * Λ * Q' - A) = 6.347597591257379e-15
```

Eigendecompositions and Matrix Powers

- Can be used to find A^t for large t (e.g. for Markov chains)
  - → P^t, i.e. $P \cdot P \cdot \ldots \cdot P$ for t times
  - → $P = Q\Lambda Q^{-1}$ then $P^t = Q\Lambda^t Q^{-1}$ where Λ^t is just the pointwise power
- Related can find matrix exponential e^A for square matrices
  - → $e^A = Qe^\Lambda Q^{-1}$ where e^Λ is just the pointwise exponential
  - → Useful for solving differential equations, e.g. $y' = Ay$ for $y(0) = y_0$ is $y(t) = e^{At}y_0$

More on Factorizations

- Plenty more used in different circumstances. Start by looking at structure
- Usually have some connection to textbook algorithms, for example LU is Gaussian elimination with pivoting and QR is Gram-Schmidt Process
- Just as shortcuts can be done with sparse matrices in textbook examples, direct sparse methods can be faster given enough sparsity
  - → But don't assume sparsity will be faster. Often slower unless matrices are big and especially sparse
  - → Dense algorithms on GPUs can be very fast because of parallelism
- Keep in mind that barring numerical roundoff issues, these are "exact" methods. They don't become more accurate with more iterations

Large Scale Systems of Equations

- Packages that solve BIG problems with “direct methods” include MUMPS, Paradiso, UMFPACK, and many others
- Sparse solvers are bread-and-butter scientific computing, so they can crush huge problems, parallelize on a cluster, etc.
- But for smaller problems they may not be ideal. Profile and test, and only if you need it.
- In Julia, the SciML package LinearSolver.jl is your best bet there, as it lets you swap out backends to profile
- On Python harder to flip between them, but scipy has many built in and many wrappers exist. Same with Matlab

Preview of Conditioning

- It will turn out that for iterative methods, a different style of algorithm, it is often necessary to multiple by a matrix to transform the problem
- The ideal transform would be the matrix's inverse, which requires a full factorization.
- But instead, you can do only part of the way towards the factorization. e.g., part of the way on gaussian elimination
- Called "Incomplete Cholesky", "Incomplete LU", etc.

Continuous Time Markov Chains

Markov Chains Transitions in in Continuous Time

- For a discrete number of states, we cannot have instantaneous transitions between states or it ceases to be measurable
- Instead: intensity of switching from state i to j as a q_{ij} where

$$\mathbb{P}\{X(t+\Delta) = j \mid X(t)\} = \begin{cases} q_{ij}\Delta + o(\Delta) & i \neq j \\ 1 + q_{ii}\Delta + o(\Delta) & i = j \end{cases}$$

- With $o(\Delta)$ is little-o notation. That is, $\lim_{\Delta \to 0} o(\Delta)/\Delta = 0$.

Intensity Matrix

- $Q_{ij} = q_{ij}$ for $i \neq j$ and $Q_{ii} = -\sum_{j \neq i} q_{ij}$
- Rows sum to 0
- For example, consider a counting process

$$Q = \begin{bmatrix} -0.1 & 0.1 & 0 & 0 & 0 & 0 \\ 0.1 & -0.2 & 0.1 & 0 & 0 & 0 \\ 0 & 0.1 & -0.2 & 0.1 & 0 & 0 \\ 0 & 0 & 0.1 & -0.2 & 0.1 & 0 \\ 0 & 0 & 0 & 0.1 & -0.2 & 0.1 \\ 0 & 0 & 0 & 0 & 0.1 & -0.1 \end{bmatrix}$$

Probability Dynamics

- The Q is the **infinitesimal generator** of the stochastic process.
- Let $\pi(t) \in \mathbb{R}^N$ with $\pi_i(t) \equiv \mathbb{P}[X_t = i \,|\, X_0]$
- Then the probability distribution evolution (Fokker-Planck or KFE), is

$$\frac{d}{dt}\pi(t) = \pi(t)Q, \qquad \text{given } \pi(0)$$

- Or, often written as $\frac{d}{dt}\pi(t) = Q^\top \cdot \pi(t)$, i.e. in terms of the "adjoint" of the linear operator Q
- A steady state is then a solution to $Q^\top \cdot \bar{\pi} = 0$

  → i.e., the $\bar{\pi}$ left-eigenvector associated with eigenvalue 0 (i.e. $\bar{\pi}Q = 0 \times \bar{\pi}$)

Setting up a Counting Process

```
alpha = 0.1
N = 6
Q = Tridiagonal(fill(alpha, N - 1),
                [-alpha; fill(-2*alpha, N - 2); -alpha],
                fill(alpha, N - 1))
Q
```

```
6×6 Tridiagonal{Float64, Vector{Float64}}:
 -0.1   0.1    ·     ·     ·     ·
  0.1  -0.2   0.1    ·     ·     ·
   ·    0.1  -0.2   0.1    ·     ·
   ·     ·    0.1  -0.2   0.1    ·
   ·     ·     ·    0.1  -0.2   0.1
   ·     ·     ·     ·    0.1  -0.1
```

Finding the Stationary Distribution

- There will always be at least one eigenvalue of 0, and the corresponding eigenvector is the stationary distribution
- Teaser: Do we really need all of the eigenvectors/eigenvalues?

```
1 Lambda, vecs = eigen(Array(Q'))
2 @show Lambda
3 vecs[:, N] ./ sum(vecs[:, N])
```

```
Lambda = [-0.3732050807568874, -0.2999999999999993,
-0.1999999999999998, -0.0999999999999995,
-0.026794919243112274, 0.0]

6-element Vector{Float64}:
 0.16666666666666657
 0.16666666666666657
 0.1666666666666667
 0.16666666666666682
 0.16666666666666685
 0.16666666666666663
```

Using the Generator in a Bellman Equation

- Let $r \in \mathbb{R}^N$ be a vector of payoffs in each state, and $\rho > 0$ a discount rate,
- Then we can use the Q generator as a simple Bellman Equation (using the Kolmogorov Backwards Equation) to find the value v in each state,

$$\rho v = r + Qv$$

- Rearranging, $(\rho I - Q)v = r$
- Teaser: can we just implement $(\rho I - Q) \cdot v$ and avoid factorizing the matrix?

Implementing the Bellman Equation

```
1 rho = 0.05
2 r = range(0.0, 10.0, length=N)
3 @show typeof(rho*I - Q)
4
5 # solve (rho * I - Q) v = r
6 v = (rho * I - Q) \ r
```

```
typeof(rho * I - Q) = LinearAlgebra.Tridiagonal{Float64,
Vector{Float64}}
```

```
6-element Vector{Float64}:
  38.15384615384615
  57.23076923076923
  84.92307692307693
 115.07692307692311
 142.76923076923077
 161.84615384615384
```