Direct Methods and Matrix Factorizations

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 core numerical linear algebra concepts
- Direct methods and matrix factorizations you'll learn:
  - → **Computational complexity**: Big-O notation and understanding what makes operations expensive
  - → **Matrix structure**: Exploiting sparsity, triangular, tridiagonal, and positive-definite structure
  - → **Factorizations**: LU, Cholesky, and eigenvalue decompositions for solving linear systems
  - → **Applications**: Continuous Time Markov Chains and Bellman equations
- These methods solve problems to machine precision but scale with matrix size
- In the next lecture, we'll see iterative methods that trade precision for scalability

Packages and Materials

- See QuantEcon Numerical Linear Algebra and associated notebooks
- Python resources:
  - → JAX SciPy Documentation - JAX implementations of SciPy functions
  - → Lineax Documentation - Linear solvers (Rader, Lyons, and Kidger 2023)

```
import jax
import jax.numpy as jnp
import jax.scipy.linalg as jla
import lineax as lx
import scipy.sparse as sp
import scipy.sparse.linalg as spla
import numpy as np
import time

# Set random seed for reproducibility
key = jax.random.PRNGKey(42)
```

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

```
print(f"machine epsilon for float64 = {jnp.finfo(jnp.float64).eps}")
print(f"1 + eps/2 == 1? {1.0 + 1.1e-16 == 1.0}")
print(f"machine epsilon for float32 = {jnp.finfo(jnp.float32).eps}")
```

```
machine epsilon for float64 = 2.220446049250313e-16
1 + eps/2 == 1? True
machine epsilon for float32 = 1.1920928955078125e-07
```

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
- We use NumPy for sparse matrix creation (JAX sparse support is experimental)

```
# Create sparse random matrix using scipy
np.random.seed(42)
A_sp = sp.random(10, 10, density=0.45, format='csr')
print(f"Non-zeros in A: {A_sp.nnz}")

# Invert (must convert to dense)
A_dense = A_sp.toarray()
invA_dense = jnp.linalg.inv(A_dense)
# Count non-zeros (threshold for numerical zeros)
invA_nnz = jnp.sum(jnp.abs(invA_dense) > 1e-10)
print(f"Non-zeros in inv(A): {invA_nnz}")
```

```
Non-zeros in A: 45
Non-zeros in inv(A): 100
```

Losing Tridiagonal Structure

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

```
N = 5
# Create tridiagonal matrix
lower = jnp.concatenate([jnp.full(N-2, 0.1), jnp.array([0.2])])
diag = jnp.full(N, 0.8)
upper = jnp.concatenate([jnp.array([0.2]), jnp.full(N-2, 0.1)])

# Build full matrix for inversion
A_tri = jnp.diag(diag) + jnp.diag(lower, -1) + jnp.diag(upper, 1)
print("Inverse of tridiagonal (all elements non-zero):")
print(jnp.linalg.inv(A_tri))
```

```
Inverse of tridiagonal (all elements non-zero):
[[ 1.2909946e+00 -3.2795697e-01  4.1666660e-02 -5.3763431e-03
   6.7204301e-04]
 [-1.6397849e-01  1.3118279e+00 -1.6666664e-01  2.1505373e-02
  -2.6881720e-03]
 [ 2.0833330e-02 -1.6666664e-01  1.2916665e+00 -1.6666664e-01
   2.0833334e-02]
 [-2.6881718e-03  2.1505374e-02 -1.6666666e-01  1.3118279e+00
  -1.6397850e-01]
 [ 6.7204307e-04 -5.3763445e-03  4.1666668e-02 -3.2795700e-01
   1.2909946e+00]]
```

Forming the Covariance and/or Gram Matrix

- Another common example is $A^T A$

```
A_sp = sp.random(20, 21, density=0.3, format='csr')
print(f"Sparsity of A: {A_sp.nnz / (20*20):.2%}")
ATA = A_sp.T @ A_sp
print(f"Sparsity of A'A: {ATA.nnz / (21*21):.2%}")
```

```
Sparsity of A: 31.50%
Sparsity of A'A: 85.94%
```

Specialized Algorithms

- Besides sparsity/storage, the real loss is you miss out on algorithms
- We'll compare dense vs. sparse vs. tridiagonal solvers

Compare Dense vs. Sparse vs. Tridiagonal

```python
N = 1000
key, subkey = jax.random.split(key)
b = jax.random.uniform(subkey, (N,))

# Create tridiagonal matrix
lower_diag = jnp.concatenate([jnp.full(N-2, 0.1), jnp.array([0.2])])
main_diag = jnp.full(N, 0.8)
upper_diag = jnp.concatenate([jnp.array([0.2]), jnp.full(N-2, 0.1)])

# Lineax tridiagonal operator (uses parallel scan, O(N))
A_tri_op = lx.TridiagonalLinearOperator(main_diag, lower_diag, upper_diag)
# Dense matrix for comparison
A_dense = jnp.diag(main_diag) + jnp.diag(lower_diag, -1) + jnp.diag(upper_diag, 1)

# Warmup both solvers (JIT compilation happens on first call)
_ = lx.linear_solve(A_tri_op, b).value.block_until_ready()
_ = jnp.linalg.solve(A_dense, b).block_until_ready()

start = time.perf_counter()
x_tri = lx.linear_solve(A_tri_op, b).value
x_tri.block_until_ready()
tri_time = time.perf_counter() - start

# Dense solve (O(N^3))
start = time.perf_counter()
x_dense = jnp.linalg.solve(A_dense, b)
x_dense.block_until_ready()
```

```
Tridiagonal (Lineax): 2.01 ms
Dense (JAX): 14.95 ms
Speedup: 7.4x
```

Key insight: Lineax uses a direct parallel scan solver (O(N)), much faster than dense solve (O(N^3))

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_small = jnp.array([1.0, 2.0, 3.0])
U = jnp.array([[1.0, 2.0, 3.0],
               [0.0, 5.0, 6.0],
               [0.0, 0.0, 9.0]])
x = jla.solve_triangular(U, b_small, lower=False)
print(f"Solution: {x}")
```

```
Solution: [0.         0.         0.33333334]
```

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. P is for partial-pivoting
- If invertible, then a $A = LU$ exists, but may not be numerically stable without pivoting
- Returns explicit matrices P, L, U (not a factorization object)

```
N_lu = 4
key, subkey = jax.random.split(key)
A = jax.random.uniform(subkey, (N_lu, N_lu))
key, subkey = jax.random.split(key)
b_lu = jax.random.uniform(subkey, (N_lu,))

# LU factorization returns explicit matrices
P, L, U = jla.lu(A)
print(f"P @ A ≈ L @ U? {jnp.allclose(P @ A, L @ U)}")
```

```
P @ A ≈ L @ U? True
```

Using LU Factorization

- To solve, use triangular solves manually or use `jnp.linalg.solve`

```python
# Manual solve using LU: solve L(Ux) = Pb
y = jla.solve_triangular(L, P @ b_lu, lower=True)
x_lu = jla.solve_triangular(U, y, lower=False)

# Direct solve for comparison
x_direct = jnp.linalg.solve(A, b_lu)

print(f"LU solution: {x_lu}")
print(f"Direct solution: {x_direct}")
print(f"Solutions match? {jnp.allclose(x_lu, x_direct)}")
```

```
LU solution: [-2.7407506   0.50105435  2.7111826   1.1074696 ]
Direct solution: [-2.7407506   0.50105435  2.7111826   1.1074696 ]
Solutions match? True
```

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 $Ly = Pb$ 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 $LY = P$, 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 positive-definite matrices
- $A = LL^T$ for lower triangular L (or $A = U^T U$ for upper triangular)

```
1 N_chol = 500
2 key, subkey = jax.random.split(key)
3 B = jax.random.uniform(subkey, (N_chol, N_chol))
4 A_pd = B.T @ B  # Easy way to generate positive definite matrix
5 print(f"A is symmetric? {jnp.allclose(A_pd, A_pd.T)}")
```

```
A is symmetric? True
```

Comparing Cholesky

```
key, subkey = jax.random.split(key)
b_chol = jax.random.uniform(subkey, (N_chol,))

# Cholesky factorization
L_chol = jla.cholesky(A_pd, lower=True)

# Solve using Cholesky
y_chol = jla.solve_triangular(L_chol, b_chol, lower=True)
x_chol = jla.solve_triangular(L_chol.T, y_chol, lower=False)

# Warmup both solvers (JIT compilation happens on first call)
_ = jnp.linalg.solve(A_pd, b_chol).block_until_ready()
_ = jla.cholesky(A_pd, lower=True).block_until_ready()
_ = jla.solve_triangular(L_chol, b_chol, lower=True).block_until_ready()
_ = jla.solve_triangular(L_chol.T, y_chol, lower=False).block_until_ready()

# Direct solve (doesn't know it's positive definite)
start = time.perf_counter()
x_direct = jnp.linalg.solve(A_pd, b_chol)
x_direct.block_until_ready()
direct_time = time.perf_counter() - start

# Cholesky solve
start = time.perf_counter()
L_chol = jla.cholesky(A_pd, lower=True)
y_chol = jla.solve_triangular(L_chol, b_chol, lower=True)
x_chol = jla.solve_triangular(L_chol.T, y_chol, lower=False)
```

```
Direct solve: 3.02 ms
Cholesky solve: 2.63 ms
Speedup: 1.1x
```

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

```jax
key, subkey = jax.random.split(key)
A_sym = jax.random.uniform(subkey, (5, 5))
A_sym = (A_sym + A_sym.T) / 2  # Make symmetric

# Eigenvalue decomposition
eigenvalues, Q = jnp.linalg.eigh(A_sym)  # eigh for symmetric/Hermitian
Lambda = jnp.diag(eigenvalues)

print(f"||Q Λ Q^-1 - A||: {jnp.linalg.norm(Q @ Lambda @ jnp.linalg.inv(Q) - A_sym):.2e}")
print(f"||Q Λ Q^T - A||: {jnp.linalg.norm(Q @ Lambda @ Q.T - A_sym):.2e}")
```

```
||Q Λ Q^-1 - A||: 7.66e-07
||Q Λ Q^T - A||: 6.90e-07
```

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

Sparse Direct Solvers: The SciPy Fallback

- **JAX limitation**: No native direct sparse solver (like UMFPACK/SuperLU)
- For sparse systems, we fall back to SciPy

```python
# Create sparse system
N_sparse = 1000
A_sparse = sp.random(N_sparse, N_sparse, density=0.01, format='csr')
A_sparse = A_sparse + sp.eye(N_sparse) * 10  # Make diagonally dominant
b_sparse = np.random.rand(N_sparse)

# Solve using SciPy's sparse solver (uses UMFPACK/SuperLU)
x_sparse = spla.spsolve(A_sparse, b_sparse)
print(f"Solved sparse system of size {N_sparse}x{N_sparse} with {A_sparse.nnz} non-zeros")
print(f"Residual: {np.linalg.norm(A_sparse @ x_sparse - b_sparse):.2e}")
```

```
Solved sparse system of size 1000x1000 with 10987 non-zeros
Residual: 1.71e-14
```

Note: For production sparse linear solves, use SciPy or interface with PETSc, not JAX

Large Scale Systems of Equations

- Packages that solve BIG problems with “direct methods” include MUMPS, Pardiso, 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.
- On Python: scipy has many built in (UMFPACK, SuperLU, etc.) 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 multiply 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 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_ctmc = 6

# Create tridiagonal Q matrix
lower_ctmc = jnp.full(N_ctmc-1, alpha)
main_ctmc = jnp.concatenate([jnp.array([-alpha]),
                             jnp.full(N_ctmc-2, -2*alpha),
                             jnp.array([-alpha])])
upper_ctmc = jnp.full(N_ctmc-1, alpha)

# Build dense matrix for display
Q = jnp.diag(main_ctmc) + jnp.diag(lower_ctmc, -1) + jnp.diag(upper_ctmc, 1)
print("Q matrix:")
print(Q)
```

```
Q matrix:
[[-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]]
```

Finding the Stationary Distribution

- There will always be at least one eigenvalue of 0, and the corresponding eigenvector is the stationary distribution
- We use dense eigenvalue decomposition here (for iterative methods, see next lecture)

```python
# Eigenvalue decomposition of Q^T
eigenvalues, eigenvectors = jnp.linalg.eig(Q.T)

# Find eigenvector corresponding to eigenvalue ≈ 0
idx = jnp.argmin(jnp.abs(eigenvalues))
pi_stationary = eigenvectors[:, idx].real
pi_stationary = pi_stationary / jnp.sum(pi_stationary)

print(f"Eigenvalues:\n{eigenvalues.real}")
print(f"\nStationary distribution:")
print(pi_stationary)
```

```
Eigenvalues:
[-3.7320518e-01 -3.0000022e-01 -2.0000000e-01 -9.9999972e-
02
 -1.0465228e-08 -2.6794920e-02]

Stationary distribution:
[0.16666669 0.16666669 0.16666669 0.16666666 0.16666664
0.16666664]
```

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, $(\rho I - Q)v = r$

Implementing the Bellman Equation

```
rho = 0.05
r = jnp.linspace(0.0, 10.0, N_ctmc)

# Solve (rho * I - Q) v = r
A_bellman = rho * jnp.eye(N_ctmc) - Q
v = jnp.linalg.solve(A_bellman, r)

print(f"Value function:")
print(v)
```

```
Value function:
[ 38.153847  57.230774  84.92308  115.076935 142.76924
161.84616 ]
```

Teaser: Can we use iterative methods to avoid forming the full matrix? See next lecture!

References

Rader, Jason, Terry Lyons, and Patrick Kidger. 2023. “Lineax: Unified Linear Solves and Linear Least-Squares in JAX and Equinox.” *AI for Science Workshop at Neural Information Processing Systems 2023*. https://arxiv.org/abs/2311.17283.