Numerical Linear Algebra with Iterative Methods

Machine Learning Fundamentals for Economists

Jesse Perla

jesse.perla@ubc.ca

University of British Columbia

Table of contents

Overview

Motivation

- Building on the previous lecture's direct methods, we now explore iterative approaches
- Iterative methods and matrix conditioning you'll learn:
  - → **Conditioning**: Why some matrices are harder to work with (condition numbers)
  - → **Stationary methods**: Jacobi iteration for diagonally dominant systems
  - → **Krylov methods**: Conjugate Gradient (CG) and NormalCG for least squares
  - → **Matrix-free operators**: Solving problems without storing the full matrix
  - → **Preconditioning**: Transforming problems to make them easier to solve
  - → **Applications**: Large-scale LLS, two-way fixed effects, CTMC value functions
- Key insight: Performance depends on geometry (conditioning), not just size
- These methods are essential for ML: optimization, regularization, and large-scale problems

Summary and Material

- See QuantEcon Krylov Methods and Matrix Conditioning
- Python resources:
  - → Lineax Documentation - JAX linear solvers (Rader, Lyons, and Kidger 2023)
  - → CoLA Documentation - Compositional Linear Algebra (Potapczynski et al. 2023)

```
import jax
import jax.numpy as jnp
import jax.random as random
import lineax as lx
import cola
import time
import matplotlib.pyplot as plt

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

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

```
epsilon = 1E-6
A2 = jnp.array([[1.0, 0.0],
                [1.0, epsilon]])
print(f"cond(A2) = {jnp.linalg.cond(A2):.2e}")
print(f"cond(A2.T) = {jnp.linalg.cond(A2.T):.2e}")
print(f"cond(inv(A2)) = {jnp.linalg.cond(jnp.linalg.inv
```

```
cond(A2) = 2.00e+06
cond(A2.T) = 2.00e+06
cond(inv(A2)) = 1.88e+06
```

Conditioning Under Matrix Products

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

```python
def lauchli(N, epsilon):
    ones_row = jnp.ones((1, N))
    eye_scaled = epsilon * jnp.eye(N)
    return jnp.vstack([ones_row, eye_scaled])

epsilon = 1E-8
L = lauchli(3, epsilon)
print(f"cond(L) = {jnp.linalg.cond(L):.2e}")
print(f"cond(L.T @ L) = {jnp.linalg.cond(L.T @ L):.2e}"
print("Matrix L:")
print(L)
```

```
cond(L) = 1.73e+08
cond(L.T @ L) = inf
Matrix L:
[[1.e+00 1.e+00 1.e+00]
 [1.e-08 0.e+00 0.e+00]
 [0.e+00 1.e-08 0.e+00]
 [0.e+00 0.e+00 1.e-08]]
```

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

Bellman Equation with CTMC Generator

- 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?

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

# Define diagonals for tridiagonal matrix Q
lower_diag = jnp.full(N-1, b)
main_diag = jnp.concatenate([jnp.array([-a]),
                             jnp.full(N-2, -(a+b)),
                             jnp.array([-b])])
upper_diag = jnp.full(N-1, a)
Q = cola.ops.Tridiagonal(lower_diag, main_diag, upper_diag)

# For direct solve, convert to dense
r = jnp.linspace(0.0, 10.0, N)
Q_dense = Q.to_dense()
A = rho * jnp.eye(N) - Q_dense
v_direct = jnp.linalg.solve(A, r)
print(f"Mean value: {jnp.mean(v_direct):.6f}")
```

```
Mean value: 101.963066
```

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,

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

See ▶ Jacobi Implementation in appendix for code example.

Structured Linear Operators

Structured and Lazy Operators

- **CoLA** (Compositional Linear Algebra) (Potapczynski et al. 2023) provides structured matrix types
- Operators can be composed lazily without materializing the result
- CoLA dispatches to appropriate algorithms based on structure (direct or iterative)
- **Example**: Diagonal + Tridiagonal

```
# Diagonal operator
D = cola.ops.Diagonal(jnp.arange(1.0, N+1))

# Compose without forming the matrix (lazy composition)
Op = Q + D

# CoLA handles the structure automatically in solves
b_test = jnp.ones(N)
x_cola = cola.solve(Op, b_test)

# Can also use eigenvalue methods
eigenvalues, eigenvectors = cola.eig(Op, k=3, which='SM')
print(f"Three smallest eigenvalues: {eigenvalues}")
```

```
Three smallest eigenvalues: [0.89475226+0.j 1.8502488 +0.j 2.850001  +0.j]
```

Benefits of Lazy Composition

- No need to materialize `Q + D` as a dense matrix
- Memory efficient for large problems
- Enables matrix-free methods at scale
- Foundation for both direct solvers and iterative methods

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

- CG method for positive-definite systems, matrix or function form

```
N_cg = 100
key, subkey = random.split(key)
A_sparse = random.uniform(subkey, (N_cg, N_cg))
key, subkey = random.split(key)
A_sparse = jnp.where(random.uniform(subkey, (N_cg, N_cg)) < 0.1, A_sparse, 0.0)
A_pd = A_sparse @ A_sparse.T + 0.5 * jnp.eye(N_cg)
key, subkey = random.split(key)
b_cg = random.uniform(subkey, (N_cg,))
x_direct = jnp.linalg.solve(A_pd, b_cg)
operator = lx.MatrixLinearOperator(A_pd, tags=lx.positive_semidefinite_tag)
solver = lx.CG(rtol=1e-5, atol=1e-5, max_steps=1000)
solution = lx.linear_solve(operator, b_cg, solver)
print(f"cond(A) = {jnp.linalg.cond(A_pd):.2e}, Iterations: {solution.stats['num_steps']}, Error: {jnp.linalg.norm(so
```

```
cond(A) = 6.32e+01, Iterations: 32, Error: 2.61e-05
```

Benchmarking: CG vs Direct Solve

```python
# Warmup both solvers (JIT compilation happens on first call)
_ = jnp.linalg.solve(A_pd, b_cg).block_until_ready()
_ = lx.linear_solve(operator, b_cg, solver).value.block_until_ready()

# Direct solve benchmark
start = time.perf_counter()
x_direct = jnp.linalg.solve(A_pd, b_cg)
x_direct.block_until_ready()  # Wait for JAX async execution
direct_time = time.perf_counter() - start

# CG solve benchmark
start = time.perf_counter()
solution = lx.linear_solve(operator, b_cg, solver)
solution.value.block_until_ready()
cg_time = time.perf_counter() - start

print(f"Direct solve: {direct_time*1000:.2f} ms")
print(f"CG solve: {cg_time*1000:.2f} ms")
print(f"Speedup: {direct_time/cg_time:.2f}x")
```

```
Direct solve: 2.51 ms
CG solve: 0.45 ms
Speedup: 5.60x
```

Key insights:

- Iterative methods scale better for large sparse systems
- Direct methods may be faster for medium/dense matrices
- Conditioning affects iteration count

Iterative Methods for LLS

- NormalCG (Rader, Lyons, and Kidger 2023) is a Krylov method for solving least squares via the normal equations

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

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

NormalCG Example

```python
M = 1000
N_lls = 10000
sigma = 0.1
key, subkey = random.split(key)
X_sparse = random.uniform(subkey, (N_lls, M))
key, subkey = random.split(key)
X_sparse = jnp.where(random.uniform(subkey, (N_lls, M)) < 0.1, X_sparse, 0.0)
key, subkey = random.split(key)
beta_true = random.uniform(subkey, (M,))
key, subkey = random.split(key)
y = X_sparse @ beta_true + sigma * random.normal(subkey, (N_lls,))
beta_direct = jnp.linalg.lstsq(X_sparse, y, rcond=None)[0]
operator = lx.MatrixLinearOperator(X_sparse)
solver = lx.NormalCG(rtol=1e-5, atol=1e-5, max_steps=1000)
solution = lx.linear_solve(operator, y, solver)
beta_normalcg = solution.value
print(f"Norm difference: {jnp.linalg.norm(beta_direct - beta_normalcg):.2e}, Iterations: {solution.stats['num_steps']
```

```
Norm difference: 1.97e-04, Iterations: 15
```

Benchmarking: Direct vs Iterative LLS

```python
# Warmup both solvers (JIT compilation happens on first call)
_ = jnp.linalg.lstsq(X_sparse, y, rcond=None)[0].block_until_ready()
_ = lx.linear_solve(operator, y, solver).value.block_until_ready()

# Benchmark direct least squares
start = time.perf_counter()
beta_direct = jnp.linalg.lstsq(X_sparse, y, rcond=None)[0]
beta_direct.block_until_ready()
direct_time = time.perf_counter() - start

# Benchmark NormalCG
start = time.perf_counter()
solution = lx.linear_solve(operator, y, solver)
solution.value.block_until_ready()
normalcg_time = time.perf_counter() - start

print(f"Direct lstsq: {direct_time*1000:.2f} ms")
print(f"NormalCG: {normalcg_time*1000:.2f} ms")
print(f"Speedup: {direct_time/normalcg_time:.2f}x")
print(f"Iterations: {solution.stats['num_steps']}")
```

```
Direct lstsq: 1283.48 ms
NormalCG: 127.66 ms
Speedup: 10.05x
Iterations: 15
```

Trade-offs:

- Overdetermined systems (N > M): iterative methods shine
- Sparse matrices: memory savings matter
- Accuracy: iterative methods controlled by tolerances

Matrix-Free LLS

- For LLS, need Xu and $X^T v$ products via `FunctionLinearOperator`
- Lineax automatically computes transposes (no manual adjoints needed!)

```
def matvec(vec):
    return X_sparse @ vec
input_structure = jax.ShapeDtypeStruct((M,), jnp.float32)
X_op = lx.FunctionLinearOperator(matvec, input_structure)
solver = lx.NormalCG(rtol=1e-5, atol=1e-5, max_steps=1000)
solution = lx.linear_solve(X_op, y, solver)
beta_matvec = solution.value
print(f"Norm diff: {jnp.linalg.norm(beta_direct - beta_matvec):.2e}, Iterations: {solution.stats['num_steps']}")
```

```
Norm diff: 7.22e-04, Iterations: 16
```

Eigenvalue Problems

Eigenvalue Example

- Steady state of CTMC is solution to $Q^{\mathsf{T}} \cdot \bar{\pi} = 0$
- The $\bar{\pi}$ left-eigenvector associated with eigenvalue 0

```
N_eig = 4
a = 0.1
b = 0.05
lower_diag = jnp.full(N_eig-1, b)
main_diag = jnp.concatenate([jnp.array([-a]), jnp.full(N_eig-2, -(a+b)), jnp.array([-b])])
upper_diag = jnp.full(N_eig-1, a)
Q_eig = cola.ops.Tridiagonal(lower_diag, main_diag, upper_diag)
Q_T = cola.ops.Tridiagonal(upper_diag, main_diag, lower_diag)
eigenvalues, eigenvectors = cola.eig(Q_T, k=1, which='SM')
lambda_min = eigenvalues[0].real
phi = eigenvectors[:, 0].real
phi = phi / jnp.sum(phi)
print(f"λ_min: {lambda_min:.2e}, Mean(φ): {jnp.mean(phi):.6f}, Q.T:\n{Q_T.to_dense()}")
```

```
λ_min: -2.50e-01, Mean(φ): 0.250000, Q.T:
[[-0.1   0.05  0.    0.  ]
 [ 0.1  -0.15  0.05  0.  ]
 [ 0.    0.1  -0.15  0.05]
 [ 0.    0.    0.1  -0.05]]
```

Implementing Matrix-Free Operator for Adjoint

```
def Q_adj_product(x):
    first = -a * x[0] + b * x[1]
    middle = a * x[:-2] - (a + b) * x[1:-1] + b * x[2:]
    last = a * x[-2] - b * x[-1]
    return jnp.concatenate([jnp.array([first]), middle, jnp.array([last])])

key, subkey = random.split(key)
x_check = random.uniform(subkey, (N_eig,))
Q_dense = Q_eig.to_dense()
error = jnp.linalg.norm(Q_adj_product(x_check) - Q_dense.T @ x_check)
print(f"Matrix-free error: {error:.2e}")
```

```
Matrix-free error: 3.73e-09
```

Solving with Matrix-Free Operator

- The `FunctionLinearOperator` wrapper adds features required for algorithms

```python
# Wrap in Lineax FunctionLinearOperator
input_structure = jax.ShapeDtypeStruct((N_eig,), jnp.float32)
Q_adj_op = lx.FunctionLinearOperator(Q_adj_product, input_structure)

# For eigenvalues, we can use CoLA with the dense version
# (CoLA's matrix-free operator support is limited for eigenvalue problems)
Q_dense_T = Q_dense.T
Q_cola = cola.ops.Dense(Q_dense_T)

# Find smallest eigenvalue
eigenvalues_mf, eigenvectors_mf = cola.eig(Q_cola, k=1, which='SM')
lambda_min_mf = eigenvalues_mf[0].real
phi_mf = eigenvectors_mf[:, 0].real
phi_mf = phi_mf / jnp.sum(phi_mf)

print(f"Smallest eigenvalue (matrix-free): {lambda_min_mf:.2e}")
print(f"Mean of eigenvector: {jnp.mean(phi_mf):.6f}")
```

```
Smallest eigenvalue (matrix-free): -2.50e-01
Mean of eigenvector: 0.250000
```

Preconditioning

Changing the Geometry

- In practice, most Krylov methods are preconditioned 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

```
N_precond = 200
key, subkey = random.split(key)
A_sparse_precond = random.uniform(subkey, (N_precond, N_precond))
key, subkey = random.split(key)
A_sparse_precond = jnp.where(random.uniform(subkey, (N_precond, N_precond)) < 0.1, A_sparse_precond, 0.0)
A_precond = A_sparse_precond @ A_sparse_precond.T + 0.5 * jnp.eye(N_precond)
key, subkey = random.split(key)
b_precond = random.uniform(subkey, (N_precond,))
operator_precond = lx.MatrixLinearOperator(A_precond, tags=lx.positive_semidefinite_tag)
solver_precond = lx.CG(rtol=1e-6, atol=1e-6, max_steps=1000)
solution_no_precond = lx.linear_solve(operator_precond, b_precond, solver_precond)
print(f"cond(A) = {jnp.linalg.cond(A_precond):.2e}, Iterations: {solution_no_precond.stats['num_steps']}")
```

```
cond(A) = 2.26e+02, Iterations: 59
```

Diagonal Preconditioner

- A simple preconditioner is the diagonal of A
- Cheap to calculate, invertible if diagonal has no zeros
- We precondition by solving $D^{-1/2}AD^{-1/2}(D^{1/2}x) = D^{-1/2}b$

```
D_inv_sqrt = 1.0 / jnp.sqrt(jnp.diag(A_precond))
P_diag = jnp.diag(D_inv_sqrt)
A_precond_system = P_diag @ A_precond @ P_diag
b_precond_system = P_diag @ b_precond
operator_precond_system = lx.MatrixLinearOperator(A_precond_system, tags=lx.positive_semidefinite_tag)
solution_precond = lx.linear_solve(operator_precond_system, b_precond_system, solver_precond)
x_precond = P_diag @ solution_precond.value
```

Diagonal Preconditioner Results

```
print(f"Iterations (with diagonal preconditioner): {solution_precond.stats['num_steps']}")
print(f"Reduction: {(1 - solution_precond.stats['num_steps']/solution_no_precond.stats['num_steps'])*100:.1f}%")
print(f"Error: {jnp.linalg.norm(A_precond @ x_precond - b_precond):.2e}")
```

```
Iterations (with diagonal preconditioner): 57
Reduction: 3.4%
Error: 1.55e-05
```

Benchmarking: Preconditioning Impact

```python
print(f"Without preconditioner: {solution_no_precond.stats['num_steps']} iterations")
print(f"With diagonal precond: {solution_precond.stats['num_steps']} iterations")
reduction_pct = (1 - solution_precond.stats['num_steps']/solution_no_precond.stats['num_steps'])*100
print(f"Reduction: {reduction_pct:.1f}%")
print(f"\nCondition numbers:")
print(f"cond(A): {jnp.linalg.cond(A_precond):.2e}")
print(f"cond(P A P): {jnp.linalg.cond(A_precond_system):.2e}")
```

```
Without preconditioner: 59 iterations
With diagonal precond: 57 iterations
Reduction: 3.4%

Condition numbers:
cond(A): 2.26e+02
cond(P A P): 2.26e+02
```

When preconditioning helps:

- Poorly conditioned systems (high κ)
- Multiple solves with same operator
- Complex problem structure
- Diagonal preconditioning reduces condition number

Incomplete Factorizations in JAX

- **Limitation**: Incomplete LU/Cholesky preconditioners are not available in the JAX ecosystem
- JAX's sparse matrix support is still experimental (`jax.experimental.sparse`)
- No mature libraries for ILU preconditioners exist for JAX (as of 2026)

Sources: JAX GitHub Discussion #18452, JAX Sparse Documentation

Other Preconditioners and Alternatives

- Diagonal preconditioners (available in Lineax)
- Algebraic multigrid methods - useful for problems with multiple scales (e.g., discretizing multiple dimensions in a statespace)

  → See **multigrid** methods

- Preconditioners for **Graph Laplacians**: approximate Cholesky decompositions and combinatorial multigrid

  → See **paper** for more

- Interface with external libraries (SciPy, PETSc) via callbacks for production use
- Julia's `IncompleteLU.jl` for comparison

Appendices

Jacobi Iteration (Educational) ▶ Back

SOR (Successive Over-Relaxation) is more complex to implement functionally in JAX due to the need for sequential updates. Use Krylov methods instead for better performance. - Educational implementation of Jacobi iteration using `jax.lax.scan`

```
# Use the CTMC example from earlier
A_jacobi = A  # From the CTMC example
b_jacobi = r
v_direct_jacobi = v_direct

# Jacobi iteration: x^{k+1} = D^{-1}(b - R x^k)
def jacobi_step(x, _):
    """Single Jacobi iteration step (functional, no mutation)"""
    D_inv = 1.0 / jnp.diag(A_jacobi)
    R = A_jacobi - jnp.diag(jnp.diag(A_jacobi))
    x_new = D_inv * (b_jacobi - R @ x)
    return x_new, None

# Run 40 iterations using scan
x0 = jnp.zeros(N)
x_jacobi, _ = jax.lax.scan(jacobi_step, x0, None, length=40)
error_jacobi = jnp.linalg.norm(x_jacobi - v_direct_jacobi, ord=jnp.inf)
print(f"Error after 40 iterations: {error_jacobi:.2e}")
```

```
Error after 40 iterations: 1.77e-03
```

References

Potapczynski, Andres, Marc Finzi, Geoff Pleiss, and Andrew Gordon Wilson. 2023. “CoLA: Exploiting Compositional Structure for Automatic and Efficient Numerical Linear Algebra.” *arXiv Preprint arXiv:2309.03060*. https://arxiv.org/abs/2309.03060.

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.