Foundations of Numerical Linear Algebra

Graduate Quantitative Economics and Datascience

Jesse Perla

jesse.perla@ubc.ca

University of British Columbia

Table of contents

Going Beyond "`reg y x, robust`"

- Data science, econometrics, and macroeconomics are built on linear algebra.
- Numerical linear algebra has all sorts of pitfalls, which become more critical as we scale up to larger problems.
- Speed differences in choosing better algorithms can be orders of magnitude.
- Crucial to know what goes on under-the-hood in Stata/R/python packages for applied work, even if you don't implement it yourself.

Extra Materials

- Material related to: QuantEcon Python, QuantEcon Data Science, Intro Quantitative Economics with Python
- **Self-study and Optional Materials:**
  - → Basics of linear algebra, matrices, norms, and linear independence
  - → Numerical optimization
  - → Systems of Equations
  - → Eigenvectors and eigenvalues
  - → Downloading and manipulating data in Python and here
  - → Introductory material on linear algebra and more
  - → Matrix decompositions and other topics

Packages

This section uses the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
import scipy
from numpy.linalg import cond, matrix_rank, norm
from scipy.linalg import inv, solve, det, eig, lu, eigvals
from scipy.linalg import solve_triangular, eigvalsh, cholesky
```

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 $O(N)$

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

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

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

Basic Linear Algebra

Norms

- Common measure of size is the Euclidean norm, or L^2 norm for $x \in \mathbb{R}^2$
- Complexity is $O(N)$, square N times then N additions

$$||x||_2 = \sqrt{\sum_{n=1}^{N} x_n^2}$$

```
x = np.array([1, 2, 3]) # Calculating different ways (in order of preference)
print(np.sqrt(sum(xval**2 for xval in x))) # manual with comprehensions
print(np.sqrt(np.sum(np.square(x)))) # broadcasts
print(norm(x)) # built-in to numpy norm(x, ord=2) alternatively
print(f"||x||_2^2 = {norm(x)**2} = {x.T @ x} = {np.dot(x, x)}")
```

```
3.7416573867739413
3.7416573867739413
3.7416573867739413
||x||_2^2 = 14.0 = 14 = 14
```

Solving Systems of Equations

- Solving $Ax = b$ for x is equivalent $A^{-1}Ax = A^{-1}b$
- Then since $A^{-1}A = I$, and $Ix = x$, we have $x = A^{-1}b$
- Careful since matrix algebra is not commutative!

```
A = np.array([[0, 2], [3, 4]]) # or ((0, 2), (3, 4))
b = np.array([2,1])  # Column vector
x = solve(A, b)  # Solve Ax = b for x
x
```

```
array([-1.,  1.])
```

Using the Inverse Directly

- Can replace the `solve` with a calculation of an inverse
- But it can be slower or less accurate than solving the system directly

```
A_inv = inv(A)
A_inv @ b # i.e, A^{-1} * b
```

```
array([-1.,  1.])
```

Linear Combinations

We can think of solving a system as finding the linear combination of columns of A that equal b

```
1 b_star = x[0] * A[:, 0] + x[1] * A[:, 1] # using x solution
2 print(f"b = {b}, b_star = {b_star}")
```

```
b = [2 1], b_star = [2. 1.]
```

Column Space and Rank

- The column space of a matrix represents all possible linear combinations of its columns.
- It forms a basis for the space of solutions when solving systems of linear equations represented by the matrix
- The rank of a matrix is the dimension of its column space

```
1 A = np.array([[0, 2], [3, 4]])
2 matrix_rank(A)
```

```
np.int64(2)
```

Hence, can solve $Ax = b$ for any $b \in \mathbb{R}^2$ since the column space is the entire space $\mathbb{R}^2$

Singular Matrices

On the other hand, note

```
A = np.array([[1, 2],
              [2, 4]])
matrix_rank(A)
```

```
np.int64(1)
```

So we can only solve $Ax = b$ for $b \propto \begin{bmatrix} 1 \\ 2 \end{bmatrix} \propto \begin{bmatrix} 2 \\ 4 \end{bmatrix}$

Checking Singularity

```
A = np.array([[1, 2], [2, 4]])
# An (expensive) way to check if A is singular is if det(A) = 0
print(det(A) == 0.0)
print(matrix_rank(A) != A.shape[0]) # or check rank
# Check before inverting or use exceptions
try:
    inv(A)
    print("Matrix is not singular (invertible).")
except np.linalg.LinAlgError:
    print("Matrix is singular (non-invertible).")
```

```
True
True
Matrix is singular (non-invertible).
```

Determinant is Not Scale Invariant

- Reminder: numerical precision in calculations makes it hard to compare to zero
- The determinate is useful but depends on the scale of the matrix
- A more robust alternative is the condition number (more next lecture)

```
1 eps, K = 1e-8, 100000
2 A = np.array([[1, 2], [1 + eps, 2 + eps]])
3 print(f"det(A)={det(A):.5g}, det(K*A)={det(K*A):.5g}")
4 print(f"cond(A)={cond(A):.5g}, cond(K*A)={cond(K*A):.5g},")
5 print(f"det(inv(A))={det(inv(A)):.5g}, cond(inv(A))={cond(inv(A)):.5g}")
```

```
det(A)=-1e-08, det(K*A)=-100
cond(A)=1e+09, cond(K*A)=1e+09,
det(inv(A))=-1e+08, cond(inv(A))=1e+09
```

Interpreting Condition Numbers

- The condition number of the matrix A is $\kappa(A) = ||A|| \cdot ||A^{-1}||$, which can be shown in terms of ratio of the largest and smallest eigenvalues
  - → $\kappa(A) = \frac{\lambda_{\max}}{\lambda_{\min}}$ for λ the eigenvalues of A. More soon!
- Crude intuition: for machine epsilon ϵ_{mach} when calculating some x
  - → The relative error, $||x - x_{\text{approx}}||/||x||$ is roughly $\kappa(A) \cdot \epsilon_{\text{mach}}$
  - → Solving $Ax = b$ when $\epsilon_{\text{mach}} = 1e^{-16}$ it amplifies errors in b, etc.
  - → if $\kappa(A) \approx 1e^{16}$ errors amplified so the scale of 100% relative error

Rules of Thumb

- Rule of thumb for standard floating points where $\epsilon_{\text{mach}} = 1e^{-16}$:
  - → $\kappa(A) \approx 1$ well-conditioned
  - → $\kappa(A) < 100$ fairly well-conditioned
  - → $\kappa(A) < 1e^{5}$ moderately ill-conditioned. Take care
  - → $\kappa(A) < 1e^{8}$ ill-conditioned and might introduce significant errors, especially in algorithms which repeatedly use the same calculations
  - → $\kappa(A) > 1e^{8}$ very ill-conditioned and likely to introduce significant errors
- Choose solution algorithms based on “numerical stability” and “conditioning” when worried
- Much more extreme with 32-bit floats such as when using GPUs.

Solving Linear Systems of Equations

Solving Systems with Multiple RHS

- Inverse is nice because you can reuse the A^{-1} to solve $Ax = b$ for many b
- However, you can do this with `solve` as well
- Or can reuse LR factorizations (discussed next)

```
A = np.array([[0, 2], [3, 4]])
B = np.array([[2,3], [1,2]])  # [2,1] and [3,2] as columns
# or: B = np.column_stack([np.array([2, 1]),np.array([3,2])])
X = solve(A, B)  # Solve AX = B for X
print(X)
print(f"Checking: A*{X[:,0]} = {A@X[:, 0]} = {B[:,0]}, column of B")
```

```
[[-1.         -1.33333333]
 [ 1.          1.5       ]]
Checking: A*[-1.  1.] = [2. 1.] = [2 1], column of B
```

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

```
A = np.array([[1, 2], [2, 4]])
P, L, U = lu(A)
print(f"L*U =\n{L @ U}")
print(f"P*A =\n{P @ A}")
```

```
L*U =
[[2. 4.]
 [1. 2.]]
P*A =
[[2. 4.]
 [1. 2.]]
```

P, U, and L

The P matrix is a permutation matrix of “pivots” the others are triangular

```
print(f"P =\n{P}")
print(f"L =\n{L}")
print(f"U =\n{U}")
```

```
P =
[[0. 1.]
 [1. 0.]]
L =
[[1.  0. ]
 [0.5 1. ]]
U =
[[2. 4.]
 [0. 0.]]
```

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)

LU for Non-Singular Matrices

```
A = np.array([[1, 2], [3, 4]])
P, L, U = lu(A)
print(f"L*U =\n{L @ U}")
print(f"P*A =\n{P @ A}")
```

```
L*U =
[[3. 4.]
 [1. 2.]]
P*A =
[[3. 4.]
 [1. 2.]]
```

L, U, P

```
1 print(f"P =\n{P}")
2 print(f"L =\n{L}")
3 print(f"U =\n{U}")
```

```
P =
[[0. 1.]
 [1. 0.]]
L =
[[1.         0.        ]
 [0.33333333 1.        ]]
U =
[[3.         4.        ]
 [0.         0.66666667]]
```

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”

Use Triangular Structure if Possible

- Triangular matrices of size N can be solved with back substitution in $O(N^2)$
- Is $O(N^2)$ good or bad? Beats, $O(N^3)$ typical of general methods

```
U = np.array([[3, 1],
              [0, 2]])
b = np.array([7,2])
solve(U,b) # works, but internally does an LU which is O(N^3)
solve_triangular(U, b, lower=False) # fast O(N^2)
```

```
array([2., 1.])
```

Symmetric Matrix Structure

Another common matrix type are symmetric, $A = A^T$

```
A = np.array([[1, 2], [2, 5]]) # also posdef, not singular
b = np.array([1,4])
# With scipy 1.11.3 check with scipy.linalg.issymmetric(A)
solve(A, b, assume_a="sym") # could also use "pos" since positive definite
```

```
array([-3.,  2.])
```

Positive Definite Matrices

- A symmetric matrix A is positive definite if $x^T Ax > 0$ for all $x \neq 0$
- Useful in many areas, such as covariance matrices. Example

```
A = np.array([[1, 2], [2, 5]])
x = np.array([0, 1]) # can't really check for all x
print(f"x^T A x = {x.T @ A @ x}")
```

```
x^T A x = 5
```

- Example of a symmetric matrix that is not positive definite

```
A = np.array([[1, 2], [2, 0]])
print(f"x^T A x = {x.T @ A @ x}")  # one counterexample is enough
```

```
x^T A x = 0
```

- We can check these with eigenvalues

Cholesky Decomposition

- For symmetric positive definite matrices: $L = U^T$,
- Called a Cholesky decomposition: $A = LL^T$ for a lower triangular matrix L.
- Equivalently, could find A = $U^T U$ for an upper triangular matrix U

```
A = np.array([[1, 2], [2, 5]])
L = cholesky(A, lower=True) # cholesky also defined for upper=True
print(L)
print(f"L*L^T =\n{L @ L.T}")
```

```
[[1. 0.]
 [2. 1.]]
L*L^T =
[[1. 2.]
 [2. 5.]]
```

Solving Positive Definite Systems

```
A = np.array([[1, 2], [2, 5]])
b = np.array([1,4])
print(solve(A, b, assume_a="pos")) # uses cholesky internally

L = cholesky(A, lower=True)
y = solve_triangular(L, b, lower=True)
x = solve_triangular(L.T, y, lower=False)
print(x)
```

```
[-3.  2.]
[-3.  2.]
```

Cholesky for Covariance Matrices

- Covariance matrices are positive-definite, semi-definite if degenerate
- Key property of Gaussian random variables:
  - → $X \sim N(\mu, \Sigma)$ for $\mu \in \mathbb{R}^N, \Sigma \in \mathbb{R}^{N \times N}$
  - → $X = \mu + AZ$ for $Z \sim N(0_N, I_N)$ where $AA^T = \Sigma$
- That is, A is the Cholesky decomposition of the covariance matrix

Matrices as Linear Transformations

- Recall: for $x \in \mathbb{R}^N$ we should think of a $f(x) = Ax$ for $A \in \mathbb{R}^{M \times N}$ as a linear transformation from $\mathbb{R}^N$ to $\mathbb{R}^M$
  - → Definition of Linear: $f(ax_1 + bx_2) = af(x_1) + bf(x_2)$ for scalar a, b
- Similarly, the $y = f(x) = Ax$ then $f^{-1}(y) = A^{-1}y$ goes from $\mathbb{R}^M$ to $\mathbb{R}^N$
  - → If the matrix is square and invertible, we can go back and forth without losing information (i.e., bijective). Otherwise we may be projected onto a lower-dimensional "manifold".

Norms and Linear Transformations

- The vector norm $||x||_2$ is an important feature in many applications
  - → Hence $||f(x)||_2 = ||Ax||_2$ frequently comes up in Quantitative Economics and Datascience
  - → e.g. linear regression is written as minimizing a vector norm

$$\min_{\beta} ||y - X\beta||_2$$

- Matrix structure or decompositions of A help us better understand the $f(x)$ mapping

Orthogonal Matrices

- A square matrix Q is **orthogonal** if: $Q^{-1} = Q^T$, and hence $Q^TQ = QQ^T = I$
  - → For orthogonal Q, $f(x) = Qx$ is interpreted as rotating x without stretching
  - → $y = f(x) = Qx$ then $f^{-1}(y) = Q^{-1}y = Q^Ty$ is rotating y back
  - → Columns are orthonormal: $Q = [q_1| \quad ... \quad |q_N]$ then
    - ↪ $q_i \cdot q_j = 0$ for $i \neq j$ and $q_i \cdot q_i = 1$
  - → Rotation means the length doesn't change: $||Qx||_2 = ||x||_2$
  - → Transformations which preserve norms are central in many applications within data science, ML, and economics - especially in high-dimensions

Eigenvalues and Eigenvectors

Eigenvalues and Eigenvectors

- For a square A, an eigenvector x and eigenvalue λ satisfy

$$Ax = \lambda x$$

- $A \in \mathbb{R}^{N \times N}$ has N eigenvalue/eigenvector pairs, possible multiplicity of λ
- Intuition: x is a direction $Ax \propto x$ and λ says how much it "stretches"

Properties of Eigenvalues and Eigenvectors

- For any eigenvector x and scalar c then $cx \propto Ax$ as well
- Symmetric matrices have real eigenvalues and orthogonal eigenvectors. i.e. $x_1 \cdot x_2 = 0$ for $x_1 \neq x_2$ eigenvectors. Complex in general
- Singular if and only if it has an eigenvalue of zero
- Positive (semi)definite if and only if all eigenvalues are strictly (weakly) positive
- Diagonal matrix has eigenvalues as its diagonal
- Triangular matrix has eigenvalues as its diagonal

Positive Definite and Eigenvalues

You cannot check $x^T Ax > 0$ for all x. Check if "stretching" is positive

```
A = np.array([[3, 1], [2, 1]])
# A_eigs = np.real(eigvals(A)) # symmetric matrices have real eigenvalues
A_eigs = eigvalsh(A) # specialized for symmetric/hermitian matrices
print(A_eigs)
is_positive_definite = np.all(A_eigs > 0)
is_positive_semi_definite = np.all(A_eigs >= 0) # or eigvals(A) >= -eps
print(f"pos-def? {is_positive_definite}")
print(f"pos-semi-def? {is_positive_semi_definite}")
```

```
[-0.23606798  4.23606798]
pos-def? False
pos-semi-def? False
```

Positive Semi-Definite Matrices **May** Have a Zero Eigenvalue

The simplest positive-semi-definite (but not posdef) matrix is

```
1 A_eigs = eigvalsh(np.array([[1, 0], [0, 0]]))
2 print(A_eigs)
3 is_positive_definite = np.all(A_eigs > 0)
4 is_positive_semi_definite = np.all(A_eigs >= 0) # or eigvals(A) >= -eps
5 print(f"pos-def? {is_positive_definite}")
6 print(f"pos-semi-def? {is_positive_semi_definite}")
```

```
[0. 1.]
pos-def? False
pos-semi-def? True
```

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

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 tools such as SVD can help with dimensionality reduction

Spectral/Eigendecomposition of Symmetric Matrix Example

```
A = np.array([[2, 1], [1, 3]])
Lambda, Q = eig(A)
print(f"eigenvectors are column-by-column in Q =\n{Q}")
print(f"eigenvalues are in Lambda = {Lambda}")
print(f"Q Lambda Q^T =\n{Q @ np.diag(np.real(Lambda)) @ Q.T}")
```

```
eigenvectors are column-by-column in Q =
[[-0.85065081 -0.52573111]
 [ 0.52573111 -0.85065081]]
eigenvalues are in Lambda = [1.38196601+0.j 3.61803399+0.j]
Q Lambda Q^T =
[[2. 1.]
 [1. 3.]]
```

Spectral Radius is Maximum Absolute Eigenvalue

- If any $\lambda \in \Lambda$ are > 1 can see this would explode
- Useful for seeing if iteration $x_{t+1} = Ax_t$ from a x_0 explodes
- The **spectral radius** of matrix A is

$$\rho(A) = \max_{\lambda \in \Lambda} |\lambda|$$