Deck 1: A Primer on Numerical Linear Algebra

Math 7870: Topics in Randomized Numerical Linear Algebra

Spring 2026

Akil Narayan

Notation

Some consistent notation we'll use:

- Non-negative integers: $n, m, p, r \in \mathbb{N}_0$
- Index sets: $[n] = \{1, 2, \ldots, n\}$
- Vectors: $\boldsymbol{v} \in \mathbb{C}^n$ (often we'll specialize to $\mathbb{R}^n$ for notational simplicity)
- Matrices: $\boldsymbol{A} \in \mathbb{C}^{m \times n}$
- Slicing vectors: Given, $S \in [n]$, then $\boldsymbol{v}_S \in \mathbb{C}^{|S|}$ is the S-sliced entries from $\boldsymbol{v}$
- Slicing matrices: Given, $S \in [n]$, then $\boldsymbol{A}_{*S} \in \mathbb{C}^{m \times |S|}$ is the S-sliced columns from $\boldsymbol{A}$
- Row slices: $\boldsymbol{A}_{T*}$ for $T \subset [m]$. The matrix $\boldsymbol{A}_{TS}$ also makes sense.
- Matrix (conjugate) transpose/adjoint, determinant, trace: $\boldsymbol{A}^* \in \mathbb{C}^{n \times m}$, $\det(\boldsymbol{A})$, $\operatorname{tr}(\boldsymbol{A})$.
- Standard inner product on vectors: $\langle \boldsymbol{v}, \boldsymbol{w} \rangle = \boldsymbol{w}^* \boldsymbol{v} \in \mathbb{C}$.
- Vectors are orthogonal (in ℓ^2) if $\langle \boldsymbol{v}, \boldsymbol{w} \rangle = 0$.

Some basic properties:

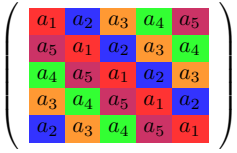
- 1-element slices produces entries: $S = \{i\}$ means $\boldsymbol{v}_S = v_i$. $T = \{j\}$ means $\boldsymbol{A}_{TS} = A_{ji}$.
- $|\det \boldsymbol{A}| = 1$ iff $\boldsymbol{A}$ has orthonormal columns.
- Norms $\|\cdot\|$ on vectors or matrices are non-negative, order-1 positively homogeneous, convex functions with trivial zero level set. (Cf. the standard ℓ^p norm.)

Matrix classifications and structure

- Square/rectangular/wide/tall
- (skew-)Hermitian
- Unitary/orthogonal
- Normal
- Diagonalizable
- Sparse
- (Orthogonal) projectors
- Circulant

Matrix structure is very useful

E.g.: A matrix $\boldsymbol{A} \in \mathbb{C}^{n\times n}$ is *circulant* if it satisfies,



$$\boldsymbol{A} = \begin{pmatrix} a_1 & a_n & a_{n-1} & \cdots & a_2 \\ a_2 & a_1 & a_n & \cdots & a_3 \\ & \ddots & \ddots & \ddots & \\ & & & & \\ a_n & a_{n-1} & a_{n-2} & \cdots & a_1 \end{pmatrix}, \quad \text{i.e.:} \begin{cases} \boldsymbol{A}_{*,i+1} &= \boldsymbol{P}\boldsymbol{A}_{*,i}, \quad i \in [n-1] \\ \boldsymbol{A}_{*,1} &= \boldsymbol{P}\boldsymbol{A}_{*,n}. \end{cases}$$

where $\boldsymbol{P}$ is the down-shift permutation matrix:

$$\boldsymbol{P} = (\boldsymbol{e}_2 \;\; \boldsymbol{e}_2 \;\; \boldsymbol{e}_3 \;\; \ldots \;\; \boldsymbol{e}_n \;\; \boldsymbol{e}_1) = \begin{pmatrix} & & & & 1 \\ 1 & & & & \\ & 1 & & & \\ & & \ddots & & \\ & & & 1 & \end{pmatrix}, \qquad \boldsymbol{P}\begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_{n-1} \\ v_n \end{pmatrix} = \begin{pmatrix} v_n \\ v_1 \\ \vdots \\ v_{n-2} \\ v_{n-1} \end{pmatrix}$$

The punchline: circulant matrices are diagonalizable by the discrete Fourier transform!
(Why is this useful...?)

Why focus on linear algebra?

The canonical reasons:

- Solving linear systems
- Vector manipulation (e.g., orthogonalizing, identifying subspaces)
- Constructing operators: projectors, discretization of differential operators

Some less obvious reasons:

- Compression, dimension reduction
- High-order optimization schemes
- Low-rank approximations
- Training for AI/LLM/neural architectures

There are "classical" applications where linear algebra is known to be useful (physics-based modeling, classical statistics, computer vision/animation).

But it's quite hard to undersell the importance of (numerical) linear algebra in modern computational applications (data science/analysis, machine learning, language models).

Computational building blocks

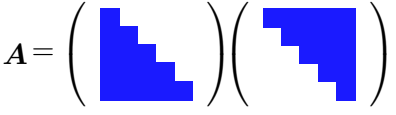
Linear algebra (LA), roughly, studies the mathematics of objects in linear (sub)spaces, which include operators.
(i.e., vectors, matrices, and linear-type operations involving these)

Numerical linear algebra (NLA), roughly, studies computational methods/algorithms that one actually uses to accomplish some linear algebraic operations.
(e.g., computing the determinant of a matrix)

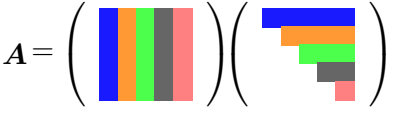
The core, classical components of NLA are matrix decompositions.

Matrix decompositions/factorizations

(LU factorization)

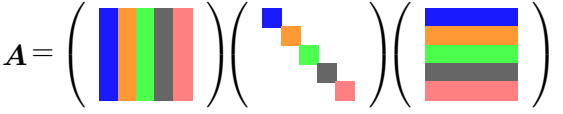


(QR factorization)



(Eigenvalue decomposition)

(Singular value decomposition)



(P)LU

Given $\boldsymbol{A} \in \mathbb{C}^{m\times n}$, its LU decomposition is,

$$\boldsymbol{A} = \boldsymbol{L}\boldsymbol{U}, \qquad \boldsymbol{L} \in \mathbb{C}^{m\times p}, \qquad \boldsymbol{U} \in \mathbb{C}^{p\times n}, \qquad p = \min\{m, n\},$$

where $\boldsymbol{L}$ is *lower triangular*, and $\boldsymbol{U}$ is *upper triangular*.

The LU decomposition need not exist for an arbitrary matrix $\boldsymbol{A}$.
(It exists iff $\det \boldsymbol{A}_{[q],[q]} \neq 0$ for all $q \in [p]$.)

A more general LU-type decomposition is the *pivoted* LU decomposition. This is one of the decompositions:

$$\begin{aligned}
\boldsymbol{A}\boldsymbol{P}_2 &= \boldsymbol{L}\boldsymbol{U}, & & \boldsymbol{P}_2 \in \mathbb{C}^{n\times n},\\
\boldsymbol{P}_1\boldsymbol{A} &= \boldsymbol{L}\boldsymbol{U}, & \boldsymbol{P}_1 \in \mathbb{C}^{m\times m}, & \\
\boldsymbol{P}_1\boldsymbol{A}\boldsymbol{P}_2 &= \boldsymbol{L}\boldsymbol{U}, & \boldsymbol{P}_1 \in \mathbb{C}^{m\times m}, & \boldsymbol{P}_2 \in \mathbb{C}^{n\times n},
\end{aligned}$$

where both $\boldsymbol{P}_1$ and $\boldsymbol{P}_2$ are row and column *permutation* matrices, respectively.
(A permutation matrix is a unitary matrix, where each row/colunn is a cardinal unit vector.)

Pivoted LU decompositions exist for any matrix.

(P)LU – in practice and uses

The computation of the LU decomposition is "easy": it's Gaussian elimination.
(This is actually how the direct computation is done on a computer.)

The permutations are chosen iteratively so that elements in pivot locations have as large a magnitude as possible.

The most popular pivoting strategy is row(-only) pivoting.

Why compute an LU decomposition?

- This is how invertible linear systems are solved: $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \longrightarrow \boldsymbol{x} = \boldsymbol{U}^{-1}\boldsymbol{L}^{-1}\boldsymbol{P}^*\boldsymbol{b}$.
- This is how "simultaneous" systems are solved (and how matrix inverses are computed).
- This is how determinants are computed: $\det \boldsymbol{A} = (\det \boldsymbol{P}_1)(\det \boldsymbol{L})(\det \boldsymbol{U})$.
- This is how (quasi-optimal) low-rank approximations are built. ("skeletonization", "empirical interpolation")
- If $\boldsymbol{A}$ is Hermitian and positive-definite, then its LU decomposition ($\boldsymbol{U} = \boldsymbol{L}^*$) is called the *Cholesky decomposition*, which is quite useful when working with these classes of matrices.

QR

Given $\boldsymbol{A} \in \mathbb{C}^{m \times n}$, its QR decomposition is,

$$\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{R}, \qquad \boldsymbol{Q} \in \mathbb{C}^{m \times m}, \qquad \boldsymbol{R} \in \mathbb{C}^{m \times n},$$

where $\boldsymbol{Q}$ is unitary ($\boldsymbol{Q}^*\boldsymbol{Q} = \boldsymbol{I}_m$), and $\boldsymbol{R}$ is upper triangular.

There is also an "economical"/"thin" QR decomposition, mainly useful when $m > n$, which truncates columns of $\boldsymbol{Q}$ corresponding to all-zero rows of $\boldsymbol{R}$.

There's also a (column-)pivoted version of QR:

$$\boldsymbol{A}\boldsymbol{P} = \boldsymbol{Q}\boldsymbol{R}, \qquad \boldsymbol{P} \in \mathbb{C}^{m \times m},$$

where $\boldsymbol{P}$ is chosen to ensure that the diagonal elements of $\boldsymbol{R}$ are non-decreasing.

QR – in practice and uses

Explicit computation of the QR decomposition is "just" orthogonalizing the column vectors of $\boldsymbol{A}$. Gram-Schmidt orthogonalization can do this – but this is a numerically unstable procedure.

"Modified" Gram-Schmidt fixes the instability, but is not really used in practice: there are procedures that employ a sequence of *unitary* transforms to compute the QR decomposition:

- Householder reflectors
- Givens rotations

This is how any modern implementation of the QR decomposition works in practice.

Why compute a QR decomposition?

- To orthogonalize vectors and/or compute orthogonal projection matrices
- This is how one could solve invertible linear systems: $\boldsymbol{Ax} = \boldsymbol{b} \longrightarrow \boldsymbol{x} = \boldsymbol{R}^{-1}\boldsymbol{Q}^*\boldsymbol{b}$.
- This is how one solves (linear) least squares problems: $\boldsymbol{Ax} = \boldsymbol{b} \longrightarrow \boldsymbol{x} = \boldsymbol{R}^{-1}(\boldsymbol{Q}_{*,[n]})^*\boldsymbol{b}$.
- This is a core ingredient in computing eigenvalues (!).

Eigenvalue decompositions

Given $\boldsymbol{A} \in \mathbb{C}^{n \times n}$, its eigenvalue decomposition is given by,

$$\boldsymbol{A} = \boldsymbol{V}\boldsymbol{\Lambda}\boldsymbol{V}^{-1}, \qquad \boldsymbol{V}, \boldsymbol{\Lambda} \in \mathbb{C}^{n \times n},$$

where $\boldsymbol{\Lambda}$ is a diagonal matrix. The diagonal elements of $\boldsymbol{\Lambda}$ are the *eigenvalues* of $\boldsymbol{A}$, and the columns of $\boldsymbol{V}$ are the corresponding *eigenvectors* of $\boldsymbol{A}$.

Square matrices having an eigenvalue decomposition are *diagonalizable*. Not all square matrices are diagonalizable. (But "most" are.)

Diagonalizable matrices are "just" diagonal matrices, when the mapping $\boldsymbol{A}$ is represented in the right coordinate system.

There is a special class of matrices for which even more is true: $\boldsymbol{A}$ is normal if $\boldsymbol{A}\boldsymbol{A}^* = \boldsymbol{A}^*\boldsymbol{A}$. A matrix is normal iff it's *unitarily diagonalizable*, i.e., is diagonalizable with $\boldsymbol{V}$ a unitary matrix.

Normal matrices are diagonal matrices, when the input/output coordinates are simply rotated/reflected.
(For example, Hermitian and skew-Hermitian matrices are normal.)

Eigenvalue decompositions – in practice and uses

Computing eigenvalues/eigenvectors is a big (+ difficult) business.
When $\boldsymbol{A}$ is normal, the business is not too bad. (Roughly speaking, the spectrum can be computed from $\boldsymbol{A}$ through a well-conditioned operation.)
The so-called "non-symmetric" eigenvalue problem is really hard – computing eigenvalues can be "arbitrarily difficult".

Here's a vague sense of how QR is used to compute eigenvalues:

- Suppose we knew $\boldsymbol{V}$.
- Compute $\boldsymbol{V} = \boldsymbol{Q}\boldsymbol{R}$.
- How could we compute the spectrum of $\boldsymbol{Q}^*\boldsymbol{A}\boldsymbol{Q}$?

Why compute eigenvalues?

- Transforming a complicated matrix to a diagonal makes it easy to understand what the matrix is "doing".
- When $\boldsymbol{A}$ is Hermitian and positive-definite, the spectrum tells us a *lot* about how to perform compression and low-rank approximation.
- We can compute singular values....

An interlude: Hermitian positive (semi)-definite matrices

Hermitian matrices $\boldsymbol{A} \in \mathbb{C}^{n \times n}$ that are *positive semi-definite*, i.e., $\boldsymbol{x}^* \boldsymbol{A} \boldsymbol{x} \geqslant \boldsymbol{0}$ for all $\boldsymbol{x}$, are so important that they deserve their own discussion.

The standard abbreviation for these matrices is "SPD". An SPD matrix $\boldsymbol{A}$ has n real eigenvalues:

$$\lambda_1 \geqslant \lambda_2 \geqslant \cdots \geqslant \lambda_n \geqslant 0.$$

These eigenvalues are informative about $\boldsymbol{A}$: $\|\boldsymbol{A}\|_2 = \lambda_1$, and $\boldsymbol{A}$ is singular iff $\lambda_n = 0$.

SPD matrices are ubiquitous, with the simplest examples being covariance matrices, (graph) Laplacian matrices, kernel matrices, Gram matrices, ...

The cone of Hermitian matrices has a useful partial ordering, the *Loewner order*:

$$\boldsymbol{A} \preceq \boldsymbol{B} \quad \Longleftrightarrow \quad \boldsymbol{B} - \boldsymbol{A} \text{ is SPD}$$

This ordering plays a nice role in matrix functional analysis: A sensible way to define a matrix function is through its spectral decomposition:

$$f : \mathbb{R} \to \mathbb{R}, \quad \boldsymbol{A} = \boldsymbol{U} \boldsymbol{\Lambda} \boldsymbol{U}^*, \quad f(\boldsymbol{A}) := \boldsymbol{U} f(\boldsymbol{\Lambda}) \boldsymbol{U}^* \quad (f(\boldsymbol{\Lambda}) \text{ diagonal, componentwise evaluation}).$$

An interesting question: are there *operator monotone* functions on SPD matrices?
I.e., if $\boldsymbol{A}, \boldsymbol{B} \succeq \boldsymbol{0}$ and $\boldsymbol{A} \preceq \boldsymbol{B}$, is $f(\boldsymbol{A}) \preceq f(\boldsymbol{B})$?

SVD

Given $\boldsymbol{A} \in \mathbb{C}^{m \times n}$, its singular value decomposition (SVD) is,

$$\boldsymbol{A} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^*, \qquad \boldsymbol{U} \in \mathbb{C}^{m \times m}, \qquad \boldsymbol{\Sigma} \in \mathbb{R}^{m \times n}, \qquad \boldsymbol{V} \in \mathbb{C}^{n \times n},$$

where both $\boldsymbol{U}$ and $\boldsymbol{V}$ are unitary, and $\boldsymbol{\Sigma}$ is diagonal with its non-negative entries arranged in non-increasing order.

The diagonal elements of $\boldsymbol{\Sigma}$ are the *singular values*, $\sigma_1, \sigma_2, \ldots, \sigma_p$, with $p = \min\{m, n\}$.

The columns of $\boldsymbol{U}$ and $\boldsymbol{V}$ are the left- and right-singular vectors of $\boldsymbol{A}$, respectively.

Sometimes, the SVD is *truncated* to remove unnecessary columns/rows from $\boldsymbol{U}/\boldsymbol{V}^*$, e.g., those corresponding to zero singluar values.

Generally, singular values and eigenvalues are unrelated.
E.g., a matrix can have all zero eigenvalues but some non-zero singular values.

Similarly, singular vectors and eigenvectors are generally unrelated.
The SVD and the eigenvalue decomposition of $\boldsymbol{A}$ coincide iff $\boldsymbol{A}$ is Hermitian and positive semi-definite.

SVD – in practice and uses

The SVD is computed through an eigenvalue decomposition of the (normal!) matrix $\boldsymbol{A}^*\boldsymbol{A}$:

$$\boldsymbol{A}\boldsymbol{A}^* = \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^*, \qquad \boldsymbol{\Lambda} = \boldsymbol{\Sigma}^2, \qquad \boldsymbol{V}^* = \boldsymbol{\Sigma}^{-1}\boldsymbol{U}^*\boldsymbol{A}$$

For simplicity, we've assumed above that $\boldsymbol{\Sigma}$ is invertible. ($\boldsymbol{\Sigma}$ is invertible iff $\boldsymbol{A}$ has full rank.)

Therefore: computing eigenvalues of Hermitian matrices is sufficient to allow us to compute the SVD.

Why compute the SVD?

- The SVD tells you "almost everything" you need to know about a matrix: its (co)range/(co)kernel, its rank, its norm for any unitarily invariant norm.
- One can solve linear systems, solve least-squares problems, compute determinants, determine matrix rank, etc.
- Truncated SVDs are *optimal* low-rank approximations of $\boldsymbol{A}$ in any unitarily invariant norm, and norms of sequence of truncated singular value are corresponding low-rank approximation errors.

Software and implementation

The core routines for computing essentially everything we've discussed is standardized through existing, nearly bulletproof software:

- (BLAS) Basic linear algebra subroutines: low-level addition/subtraction of vectors, scalar multiplication, matrix-vector and matrix-matrix multiplication.
- (LAPACK) The linear algebra package: eigenvalues, linear solvers, matrix factorization (LU, QR, Cholesky, EVD, SVD)

These are *modular* routines: LAPACK uses BLAS for its building block.
E.g., GPU-enabled implementations of dense linear algebra factorizations can, "in principle", be implemented by recoding just the BLAS portion without changing LAPACK. (Cf. cuBLAS)

This is all great: what's the problem?

All of the above is *dense* linear algebra: matrices are stored and manipulated as explicit arrays, where each array entry is explicitly stored and arithmetically exercised.

Modern practice and bottlenecks

The problem: We always want our hammer to smack bigger, tougher nails.

Using these established routines:

- For $n \times n$ matrices, matmat multiplications have $\mathcal{O}(n^3)$ complexity, matvecs are $\mathcal{O}(n^2)$.
- For $\boldsymbol{A} \in \mathbb{C}^{m \times n}$, $m \geqslant n$, computing its LU, QR, or SVD factorizations requires $\mathcal{O}(mn^2)$ effort, with $\mathcal{O}(mn)$ memory.
- For $\boldsymbol{A} \in \mathbb{C}^{n \times n}$, solving a corresponding linear system requires $\mathcal{O}(n^3)$ effort, and $\mathcal{O}(n^2)$ memory.

These are (effectively) optimal complexities for these operations.

For context:

- Datasets often feature $m, n \sim 10^6$.
- ML architectures routinely have 10^6 parameters. (We need to compute matvecs with these parameters, store/use gradients, or maybe even Hessians?)
- In physics-based models, we'd *love* to routinely solve linear systems with $n \gg 10^{10}$.
- Even if matrices/vectors are *sparse* and you can get away with storing them with reasonable memory requirements, many intermediate NLA quantities are dense.
- In general, matrices/vectors do not have special structure/sparsity that we can exploit.

This is the problem: the scale of modern problems defy naive usage of dense NLA procedures.

The potential, pitfalls, and promise of *randomized* NLA

PSA: There are no free lunches here.
If we are willing to give up nothing, we cannot really gain much more.

But there are some things we might be willing to give up:

- Maybe we only need an answer to low precision
- Maybe we only need some part of an answer
- Maybe we're willing to cope with a possibility, hopefully minute, of algorithmic failure

It is here where *randomized* methods hold promise:

- I can get "close" to the answer by randomly approximating the problem
- I can get "part" of the answer by randomly compressing the problem
- Randomness entails some possibility of failure

The foundation of randomized NLA methods is laid in probability and statistics....