# Numerical methods for solving linear systems of equations

Here are the least-squares problems we have talked about so far:

| | |
|---|---|
| Pseudo-inverse when $\boldsymbol{A}$ has full column rank | $\hat{\boldsymbol{x}} = (\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A})^{-1}\boldsymbol{A}^{\mathrm{T}}\boldsymbol{y}$ |
| Pseudo-inverse when $\boldsymbol{A}$ has full row rank | $\hat{\boldsymbol{x}} = \boldsymbol{A}^{\mathrm{T}}(\boldsymbol{A}\boldsymbol{A}^{\mathrm{T}})^{-1}\boldsymbol{y}$ |
| Ridge Regression / Tikhonov regularization | $\hat{\boldsymbol{x}} = (\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A} + \delta\mathbf{I})^{-1}\boldsymbol{A}^{\mathrm{T}}\boldsymbol{y}$ |
| Kernel Regression | $\hat{\boldsymbol{\alpha}} = (\boldsymbol{K} + \delta\mathbf{I})^{-1}\boldsymbol{y}$ |

We also encountered this "ridge regression in Hilbert space" problem at one point:

$$\underset{\boldsymbol{x}}{\text{minimize}} \ \|\boldsymbol{y} - \boldsymbol{A}\boldsymbol{x}\|_2^2 + \delta \boldsymbol{x}^{\mathrm{T}}\boldsymbol{G}\boldsymbol{x},$$

which has solution

$$\hat{\boldsymbol{x}} = (\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A} + \delta\boldsymbol{G})^{-1}\boldsymbol{A}^{\mathrm{T}}\boldsymbol{y}.$$

Here is one that we did not talk about, but it is an easy extension:

### Constrained least-squares

$$\underset{\boldsymbol{x}}{\text{minimize}} \, \|\boldsymbol{y} - \boldsymbol{A}\boldsymbol{x}\|_2^2 \quad \text{subject to} \quad \boldsymbol{x} = \boldsymbol{B}\boldsymbol{\alpha} \quad \text{for some } \boldsymbol{\alpha},$$

has solution

$$\hat{\boldsymbol{x}} = \boldsymbol{B}(\boldsymbol{B}^{\mathrm{T}}\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}\boldsymbol{B})^{-1}\boldsymbol{B}^{\mathrm{T}}\boldsymbol{A}^{\mathrm{T}}\boldsymbol{y}.$$

Each of the problems on the previous page involves solving a system of **symmetric positive definite** system of equations.

There are many ways to solve general systems of equations. Most general methods revolve around factoring the matrix $\boldsymbol{A}$ into a series of systems that are much easier to solve — doing this allows us to re-use our work if we have multiple right-hand sides.

We will start with a (very) brief overview of how to solve general systems of equations using explicit matrix computations. If you want to read more about this, the classic reference is:

G. H. Golub and C. F. van Loan, *Matrix Computations*, now in its 4th edition (2012).

First, to set some context, let's look at some particular types of systems which are "easy" to solve. In all of the examples below, $\boldsymbol{A}$ is an invertible $N \times N$ matrix.

**Diagonal systems.** If

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & 0 & \cdots & \\ 0 & a_{22} & & \\ & & \ddots & \\ & & & a_{NN} \end{bmatrix}$$

with $a_{nn} \neq 0$, then solving $\boldsymbol{Ax} = \boldsymbol{b}$ for a given $\boldsymbol{b}$ is easy; simply take

$$\hat{x}[n] = b[n]/a_{nn}.$$

This is of course very efficient computationally; we can compute $\hat{\boldsymbol{x}}$ in $O(N)$ time.

**Example.** Solve

$$\begin{bmatrix} 3 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 4 \end{bmatrix} \begin{bmatrix} x[1] \\ x[2] \\ x[3] \end{bmatrix} = \begin{bmatrix} 0.5 \\ 14 \\ 7 \end{bmatrix}$$

**Orthogonal systems.** If the columns (or equivalently the rows) of $\boldsymbol{A}$ are orthonormal, $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A} = \mathbf{I}$, then $\boldsymbol{A}^{-1} = \boldsymbol{A}^{\mathrm{T}}$, and we can solve $\boldsymbol{Ax} = \boldsymbol{b}$ with a single matrix-vector multiply:

$$\hat{\boldsymbol{x}} = \boldsymbol{A}^{\mathrm{T}}\boldsymbol{b}.$$

In general, the cost of this matrix-vector multiply is $O(N^2)$.

**Example:** Solve

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1/\sqrt{2} & 1/\sqrt{2} \\ 0 & -1/\sqrt{2} & 1/\sqrt{2} \end{bmatrix} \begin{bmatrix} x[1] \\ x[2] \\ x[3] \end{bmatrix} = \begin{bmatrix} 3 \\ 0 \\ 1 \end{bmatrix}$$

**Triangular systems.** If $\boldsymbol{A}$ is lower triangular in that all of the terms above its main diagonal are zero,

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & 0 & 0 & & \\ a_{21} & a_{22} & 0 & & \\ a_{31} & a_{32} & a_{33} & & \\ \vdots & & & \ddots & \\ a_{N1} & a_{N2} & \cdots & & a_{NN} \end{bmatrix},$$

with $a_{nn} \neq 0$, then given $\boldsymbol{b}$, we can solve $\boldsymbol{Ax} = \boldsymbol{b}$ using *forward substitution*:

$$\begin{aligned} \hat{x}[1] &= b[1]/a_{11} \\ \hat{x}[2] &= (b[2] - a_{21}\hat{x}[1])/a_{22} \\ &\vdots \\ \hat{x}[N] &= \left( b[N] - \sum_{n=1}^{N-1} a_{Nn}\hat{x}[n] \right) /a_{NN}. \end{aligned}$$

The total cost of this is about the same as a vector-matrix multiply, $O(N^2)$. If $\boldsymbol{A}$ is upper triangular,

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1N} \\ 0 & a_{22} & a_{23} & & a_{2N} \\ 0 & 0 & a_{33} & & a_{3N} \\ \vdots & & & \ddots & \\ 0 & & & & a_{NN} \end{bmatrix},$$

then given $\boldsymbol{y}$ we can solve $\boldsymbol{Ax} = \boldsymbol{y}$ using *backward substitution*. (Write it down at home!)

**Example:** Solve

$$\begin{bmatrix} 2 & 0 & 0 \\ -1 & 3 & 0 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x[1] \\ x[2] \\ x[3] \end{bmatrix} = \begin{bmatrix} 14 \\ -2 \\ 1 \end{bmatrix}$$

# Matrix factorization

The general strategy for solving a system of equations is to **factor** the matrix $\boldsymbol{A}$ into multiple components, each of which has one of the structures above. These factorizations are not significantly more expensive than solving a system using Gaussian elimination, and once they are performed, solving another system $\boldsymbol{Ax} = \boldsymbol{b}$ with the same $\boldsymbol{A}$ (but different $\boldsymbol{b}$) is fast.

## $\boldsymbol{LU}$ factorization

Every $N \times N$ matrix $\boldsymbol{A}$ can be written as a product $\boldsymbol{A} = \boldsymbol{LU}$, where $\boldsymbol{L}$ is lower diagonal, and $\boldsymbol{U}$ is upper diagonal[1]. The decomposition is in general not unique, but if we restrict the diagonal entries of $\boldsymbol{L}$ (or $\boldsymbol{U}$) to be 1 (or anything $\neq 0$), then it becomes unique (when $\boldsymbol{A}$ is invertible).

**Example:**

$$\begin{bmatrix} 2 & 1 & -1 \\ -3 & -1 & 2 \\ -2 & 1 & 2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ -1.5 & 1 & 0 \\ -1 & 4 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & -1 \\ 0 & 0.5 & -0.5 \\ 0 & 0 & -1 \end{bmatrix}$$

$$\boldsymbol{A} \qquad = \qquad \boldsymbol{L} \qquad\qquad \boldsymbol{U}$$

If $\boldsymbol{A}$ is invertible, then both $\boldsymbol{L}$ and $\boldsymbol{U}$ are invertible. So given $\boldsymbol{Ax} =$

[1] Actually, this is not technically true in exactly the way we have stated it. A more precise statement would be that we can factor into $\boldsymbol{LU}$ for some permutation of the rows of $\boldsymbol{A}$; that is, there exists a permutation matrix $\boldsymbol{P}$ such that $\boldsymbol{PA} = \boldsymbol{LU}$. This is more of a technical concern, though, and doesn't change the story we are telling here.

$\boldsymbol{b}$, we can solve for $\boldsymbol{x}$ as follows:

$$\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b} = \boldsymbol{U}^{-1}\boldsymbol{L}^{-1}\boldsymbol{b}.$$

More explicitly, we solve for $\boldsymbol{w}$ in $\boldsymbol{L}\boldsymbol{w} = \boldsymbol{b}$, then solve for $\boldsymbol{x}$ in $\boldsymbol{U}\boldsymbol{x} = \boldsymbol{w}$. As we have argued above, the cost of solving each of these systems of equations with the factorization in place is $O(N^2)$ (as opposed to $O(N^3)$).

Computing the $\boldsymbol{LU}$ factorization is basically the same as recording all of your work while you are doing Gaussian elimination. We start with the original matrix, perform row operations on it until it is in upper triangular form. Each of the row operations corresponds to a lower-diagonal matrix with one off diagonal term, and their product will also be lower diagonal.

**Example:** Start with the original matrix

$$\boldsymbol{A} = \begin{bmatrix} 2 & 1 & -1 \\ -3 & -1 & 2 \\ -2 & 1 & 2 \end{bmatrix}$$

Eliminate the lower-left term. In this case we can add the first row to the third row, i.e.

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}}_{\boldsymbol{L}_1} \underbrace{\begin{bmatrix} 2 & 1 & -1 \\ -3 & -1 & 2 \\ -2 & 1 & 2 \end{bmatrix}}_{\boldsymbol{A}} = \underbrace{\begin{bmatrix} 2 & 1 & -1 \\ -3 & -1 & 2 \\ 0 & 2 & 1 \end{bmatrix}}_{\boldsymbol{A}_1}$$

Eliminate the term in (row,column) $= (2, 1)$ by adding $3/2$ the top row to the second row:

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 1.5 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\boldsymbol{L}_2} \underbrace{\begin{bmatrix} 2 & 1 & -1 \\ -3 & -1 & 2 \\ 0 & 2 & 1 \end{bmatrix}}_{\boldsymbol{A}_1} = \underbrace{\begin{bmatrix} 2 & 1 & -1 \\ 0 & 0.5 & 0.5 \\ 0 & 2 & 1 \end{bmatrix}}_{\boldsymbol{A}_2}$$

Finally, we eliminate $(3,2)$ by subtracting 4 times the second row from the third row:

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -4 & 1 \end{bmatrix}}_{\boldsymbol{L}_3} \underbrace{\begin{bmatrix} 2 & 1 & -1 \\ 0 & 0.5 & 0.5 \\ 0 & 2 & 1 \end{bmatrix}}_{\boldsymbol{A}_2} = \underbrace{\begin{bmatrix} 2 & 1 & -1 \\ 0 & 0.5 & 0.5 \\ 0 & 0 & -1 \end{bmatrix}}_{\boldsymbol{U}}.$$

So we have:

$$\boldsymbol{L}_3\boldsymbol{L}_2\boldsymbol{L}_1\boldsymbol{A} = \boldsymbol{U},$$

where $\boldsymbol{U}$ is upper triangular and the $\boldsymbol{L}_i$ are all lower triangular with 1 along the diagonal and exactly 1 non-zero off-diagonal term. Using the facts that

1. the inverses of $\boldsymbol{L}_i$ are also lower triangular with 1 down the diagonal and exactly one non-zero off-diagonal term (show this at home!)[2], and
2. the product of two lower triangular matrices with 1 down the diagonal is again lower-triangular with 1 down the diagonal (show this at home!),

we have

$$\boldsymbol{A} = (\boldsymbol{L}_1^{-1}\boldsymbol{L}_2^{-1}\boldsymbol{L}_3^{-1})\boldsymbol{U} = \boldsymbol{L}\boldsymbol{U},$$

where $\boldsymbol{L}$ is lower diagonal with 1 down the diagonal. In the example

[2] You need to show the following. If $\boldsymbol{e}_i$ and $\boldsymbol{e}_j$ are standard unit vectors (i.e. columns of the identity), then $(\mathbf{I} + \alpha\boldsymbol{e}_i\boldsymbol{e}_j^{\mathrm{T}})^{-1} = \mathbf{I} - \alpha\boldsymbol{e}_i\boldsymbol{e}_j^{\mathrm{T}}$.

above, we have

$$\boldsymbol{L} = \boldsymbol{L}_1^{-1}\boldsymbol{L}_2^{-1}\boldsymbol{L}_3^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ -1.5 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 4 & 1 \end{bmatrix}$$
$$= \begin{bmatrix} 1 & 0 & 0 \\ -1.5 & 1 & 0 \\ -1 & 4 & 1 \end{bmatrix}$$

If we count up the operations above, we have to eliminate $(N-1)N/2$ terms, each at a cost of $O(N)$, so computing the $\boldsymbol{LU}$ factorization is $O(N^3)$ (same as using Gaussian elimination to solve $\boldsymbol{Ax} = \boldsymbol{b}_0$ for a particular $\boldsymbol{b}_0$).

## Cholesky factorization

When $\boldsymbol{A}$ is symmetric positive definite, we can take the lower- and upper-triangular factors to be transposes of one another:

$$\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^{\mathrm{T}}.$$

In this case, $\boldsymbol{L}$ will not have 1s along the diagonal. Algorithms used to compute the Cholesky factorization are similar to those that compute $\boldsymbol{LU}$ factorizations. The basic idea is that you can use elimination to find a lower-triangular matrix $\boldsymbol{R}_1$ that eliminates all but the first entry in the first column of $\boldsymbol{A}$:

$$\boldsymbol{R}_1 \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ \vdots & & \ddots & \\ a_{N1} & a_{N2} & \cdots & a_{NN} \end{bmatrix} = \begin{bmatrix} \sqrt{a_{11}} & a'_{12} & \cdots & a'_{1N} \\ 0 & a'_{22} & \cdots & a'_{2N} \\ \vdots & & \ddots & \\ 0 & a'_{N2} & \cdots & a'_{NN} \end{bmatrix}.$$

Since $\boldsymbol{A}$ is symmetric, we can do the same elimination on the columns to get

$$\boldsymbol{R}_1\boldsymbol{A}\boldsymbol{R}_1^{\mathrm{T}} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & a''_{22} & \cdots & a''_{2N} \\ \vdots & & \ddots & \\ 0 & a''_{N2} & \cdots & a''_{NN} \end{bmatrix} = \begin{bmatrix} 1 & \mathbf{0}^{\mathrm{T}} \\ \mathbf{0} & \boldsymbol{A}_1 \end{bmatrix}$$

It is easy to see that $\boldsymbol{A}_1$ must also be symmetric (and positive definite), so we continue by eliminating all but the first entry in its first column using another lower-triangular matrix $\boldsymbol{R}_2$, then doing the same to the columns to get

$$\boldsymbol{R}_2\boldsymbol{R}_1\boldsymbol{A}\boldsymbol{R}_1^{\mathrm{T}}\boldsymbol{R}_2^{\mathrm{T}} = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{A}_2 \end{bmatrix}.$$

After $N$ such iterations, we have

$$\boldsymbol{R}_N \cdots \boldsymbol{R}_1\boldsymbol{A}\boldsymbol{R}_1^{\mathrm{T}} \cdots \boldsymbol{R}_N^{\mathrm{T}} = \boldsymbol{R}\boldsymbol{A}\boldsymbol{R}^{\mathrm{T}} = \mathbf{I}.$$

Since the product of two lower-triangular matrices is again lower triangular (show this at home!), $\boldsymbol{R}$ is again lower-triangular. Since the inverse of a lower-triangular matrix is again lower triangular (show this at home!), we can take

$$\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^{\mathrm{T}}, \quad \text{with} \quad \boldsymbol{L} = \boldsymbol{R}^{-1}.$$

The Cholesky decomposition is unique. It can actually be computed slightly faster than a general $\boldsymbol{LU}$ decomposition, and is easier to stabilize.

**Example.**

$$\begin{bmatrix} 6 & 3 & 0 \\ 3 & 4 & 1 \\ 0 & 1 & 3 \end{bmatrix} = \begin{bmatrix} 2.4495 & 0 & 0 \\ 1.2247 & 1.5811 & 0 \\ 0 & 0.6325 & 1.6125 \end{bmatrix} \begin{bmatrix} 2.4495 & 1.2247 & 0 \\ 0 & 1.5811 & 0.6325 \\ 0 & 0 & 1.6125 \end{bmatrix}$$

## $QR$ factorization

The $\boldsymbol{QR}$ decomposition factors $\boldsymbol{A}$ as

$$\boldsymbol{A} = \boldsymbol{QR},$$

where $\boldsymbol{Q}$ is orthogonal, and $\boldsymbol{R}$ is upper triangular. It can be computed by running (a stabilized version of) Gram-Schmidt on the columns of $\boldsymbol{A}$; its computational complexity is again $O(N^3)$. Once we have it in hand, solving $\boldsymbol{Ax} = \boldsymbol{b}$ has the cost of solving an orthogonal system ($O(N^2)$) and a triangular system ($O(N^2)$).

We will explore this connection more on the next homework.

Like all of the other decompositions in this section, Computing a $\boldsymbol{QR}$ decomposition costs $O(N^3)$, but once it is in place, we can solve $\boldsymbol{Ax} = \boldsymbol{b}$ using $\boldsymbol{x} = \boldsymbol{R}^{-1}\boldsymbol{Q}^{\mathrm{T}}\boldsymbol{b}$ in $O(N^2)$ time.

## SVD and Eigenvalue decompositions

We have already become familiar with the singular value decomposition for general matrices:

$$\boldsymbol{A} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^{\mathrm{T}}.$$

When $\boldsymbol{A}$ is square and invertible ($\mathrm{rank}(\boldsymbol{A}) = N$), then all of $\boldsymbol{U}, \boldsymbol{\Sigma}$, and $\boldsymbol{V}$ are $N \times N$ and $\boldsymbol{U}\boldsymbol{U}^{\mathrm{T}} = \boldsymbol{U}^{\mathrm{T}}\boldsymbol{U} = \boldsymbol{V}\boldsymbol{V}^{\mathrm{T}} = \boldsymbol{V}^{\mathrm{T}}\boldsymbol{V} = \mathbf{I}$. As we have seen, we can solve $\boldsymbol{Ax} = \boldsymbol{b}$ with

$$\boldsymbol{x} = \boldsymbol{V}\boldsymbol{\Sigma}^{-1}\boldsymbol{U}^{\mathrm{T}}\boldsymbol{b}.$$

We can see that with the SVD in place, the cost of solving a system is $O(N^2)$.

For symmetric $\boldsymbol{A}$, we can write

$$\boldsymbol{A} = \boldsymbol{V}\boldsymbol{\Lambda}\boldsymbol{V}^{\mathrm{T}},$$

and then $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ is solved with $\boldsymbol{x} = \boldsymbol{V}\boldsymbol{\Lambda}^{-1}\boldsymbol{V}^{\mathrm{T}}\boldsymbol{b}$.

Both of these decompositions represent a matrix as orthogonal-diagonal-orthogonal. Computing either costs $O(N^3)$, and they are slightly more expensive than the $\boldsymbol{QR}$ and $\boldsymbol{LU}$ decompositions above. In fact, computing a $\boldsymbol{QR}$ decomposition is often used as a stepping stone to computing the SVD or eigenvalue decomposition.

## Computing eigenvalue decompositions of symmetric matrices

For $N \times N$ symmetric positive semi-definite $\boldsymbol{A}$, there are many ways to compute the eigenvalue decomposition $\boldsymbol{A} = \boldsymbol{V}\boldsymbol{\Lambda}\boldsymbol{V}^{\mathrm{T}}$. We discuss here one particular technique, popular for its stability, flexibility, and speed, based on *power iterations.*

### Power iterations for computing $\boldsymbol{v}_1$

To start, let's consider the simpler problem of computing the largest eigenvalue $\lambda_1$ and corresponding eigenvector $\boldsymbol{v}_1$ of $\boldsymbol{A}$. We do this with the following iteration.

Let $\boldsymbol{q}_0$ be an arbitrary vector in $\mathbb{R}^N$ with unit norm, $\|\boldsymbol{q}_0\|_2 = 1$.

Then for $k = 1, 2, \ldots$ compute

$$\begin{aligned} \boldsymbol{z}_k &= \boldsymbol{A}\boldsymbol{q}_{k-1} \\ \boldsymbol{q}_k &= \frac{\boldsymbol{z}_k}{\|\boldsymbol{z}_k\|_2} \\ \gamma_k &= \boldsymbol{q}_k^{\mathrm{T}} \boldsymbol{A}\boldsymbol{q}_k \end{aligned}$$

Then, as long as $\boldsymbol{q}_0$ is not orthogonal to $\boldsymbol{v}_1$, $\langle \boldsymbol{q}_0, \boldsymbol{v}_1 \rangle \neq 0$, and $\lambda_1 > \lambda_2$, as $k$ gets large

$$\boldsymbol{q}_k \to \boldsymbol{v}_1 \quad \text{and} \quad \gamma_k \to \lambda_1.$$

A detailed proof of this, including rates of convergence, can be found in Section 8.2 of the Golub/van Loan book. But we can see roughly why it works with a simple calculation.

The vector $\boldsymbol{q}_k$ above can be written as

$$\boldsymbol{q}_k = \frac{\boldsymbol{A}^k \boldsymbol{q}_0}{\|\boldsymbol{A}^k \boldsymbol{q}_0\|_2}.$$

Since the eigenvectors of $\boldsymbol{A}$, $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_N$ form an orthobasis for $\mathbb{R}^N$, we can write

$$\boldsymbol{q}_0 = \alpha_1 \boldsymbol{v}_1 + \alpha_2 \boldsymbol{v}_2 + \cdots + \alpha_N \boldsymbol{v}_N,$$

for $\alpha_n = \langle \boldsymbol{q}_0, \boldsymbol{v}_n \rangle$. Then the expression for $\boldsymbol{q}_k$ above becomes

$$\boldsymbol{q}_k = \frac{\alpha_1 \lambda_1^k \boldsymbol{v}_1 + \alpha_2 \lambda_2^k \boldsymbol{v}_2 + \cdots + \alpha_N \lambda_N^k \boldsymbol{v}_N}{\sqrt{\alpha_1^2 \lambda_1^{2k} + \alpha_2^2 \lambda_2^{2k} + \cdots + \alpha_N^2 \lambda_N^{2k}}}.$$

If $\alpha_1 \neq 0$ and $\lambda_1 > \lambda_2 \geq \cdots \geq \lambda_N$, then as $k$ gets large, the first term in each of the sums above will dominate. Thus for large $k$,

$$\boldsymbol{q}_k \approx \frac{\alpha_1 \lambda_1^k \boldsymbol{v}_1}{\sqrt{\alpha_1^2 \lambda_1^{2k}}} = \boldsymbol{v}_1.$$

Since $\boldsymbol{v}_1^{\mathrm{T}} \boldsymbol{A} \boldsymbol{v}_1 = \lambda_1$ and $\boldsymbol{q}_k \approx \boldsymbol{v}_1$, we also have that $\boldsymbol{q}_k^{\mathrm{T}} \boldsymbol{A} \boldsymbol{q}_k \approx \lambda_1$.

## QR iterations

Now consider the problem of computing all the eigenvectors and eigenvalues of $\boldsymbol{A}$. We might be tempted to extend the the power method by starting with an entire orthobasis[3] $\boldsymbol{Q}_0$, then take

$$\boldsymbol{Z}_k = \boldsymbol{A}\boldsymbol{Q}_{k-1},$$

and then renormalize the columns of $\boldsymbol{Z}_k$ to get $\boldsymbol{Q}_k$. The problem with this is that all of the columns of $\boldsymbol{Z}_k$ will converge to $\boldsymbol{v}_1$ — this is just running the power method with $N$ different starting points.

What we do instead is "orthonormalize" the columns of $\boldsymbol{Z}_k$: we make them orthogonal to each other at every iteration as well as unit norm. This gives us the following iteration:

Let $\boldsymbol{Q}_0$ be any orthonormal matrix. For $k = 1, 2, \ldots,$ take

$$\boldsymbol{Z}_k = \boldsymbol{A}\boldsymbol{Q}_{k-1} \tag{1}$$

$$[\boldsymbol{Q}_k, \boldsymbol{R}_k] = \mathrm{qr}(\boldsymbol{Z}_k) \qquad (\text{so } \boldsymbol{Z}_k = \boldsymbol{Q}_k\boldsymbol{R}_k). \tag{2}$$

Using arguments not too different than for the power method, you can show (again, see Golub/van Loan Section 8.2 for details) that

$$\boldsymbol{Q}_k \to \boldsymbol{V}, \quad \text{and} \quad \boldsymbol{\Gamma}_k = \boldsymbol{Q}_k^{\mathrm{T}}\boldsymbol{A}\boldsymbol{Q}_k \to \boldsymbol{\Lambda}.$$

A more popular way to state the iteration (1)–(2) above, and the one you will see in almost every textbook on numerical linear algebra, is the following.

Set $\boldsymbol{\Gamma}_0 = \boldsymbol{A}$. Then for $k = 1, 2, \ldots,$ take

$$[\boldsymbol{U}_k, \boldsymbol{R}_k] = \mathrm{qr}(\boldsymbol{\Gamma}_{k-1}) \qquad (\text{so } \boldsymbol{\Gamma}_{k-1} = \boldsymbol{U}_k\boldsymbol{R}_k) \tag{3}$$

$$\boldsymbol{\Gamma}_k = \boldsymbol{R}_k\boldsymbol{U}_k \tag{4}$$

[3] So $\boldsymbol{Q}_0$ is $N \times N$ and satisfies $\boldsymbol{Q}_0^{\mathrm{T}}\boldsymbol{Q}_0 = \mathbf{I}$.

So at each iteration, we are computing a QR factorization, then reversing it (cute!). This gives us the relation

$$\boldsymbol{R}_{k-1}\boldsymbol{U}_{k-1} = \boldsymbol{U}_k\boldsymbol{R}_k.$$

To see the relationship between version 1 in (1)–(2) and version 2 in (3)–(4), notice that if we initialize version 1 with $\boldsymbol{Q}_0 = \mathbf{I}$, then

$$\begin{aligned}
\boldsymbol{A} = \boldsymbol{Q}_1\boldsymbol{R}_1 &\Rightarrow \boldsymbol{Q}_1 = \boldsymbol{A}\boldsymbol{R}_1^{-1} \\
\boldsymbol{A}\boldsymbol{Q}_1 = \boldsymbol{Q}_2\boldsymbol{R}_2 &\Rightarrow \boldsymbol{Q}_2 = \boldsymbol{A}^2\boldsymbol{R}_1^{-1}\boldsymbol{R}_2^{-1} \\
&\vdots \\
\boldsymbol{A}\boldsymbol{Q}_{k-1} = \boldsymbol{Q}_k\boldsymbol{R}_k &\Rightarrow \boldsymbol{Q}_k = \boldsymbol{A}^k\boldsymbol{R}_1^{-1}\boldsymbol{R}_2^{-1}\cdots\boldsymbol{R}_k^{-1}.
\end{aligned}$$

In version 2, we have

$$\begin{aligned}
\boldsymbol{A} &= \boldsymbol{U}_1\boldsymbol{R}_1 \\
\boldsymbol{A}^2 &= \boldsymbol{U}_1\boldsymbol{R}_1\boldsymbol{U}_1\boldsymbol{R}_1 = \boldsymbol{U}_1\boldsymbol{U}_2\boldsymbol{R}_2\boldsymbol{R}_1 \\
\boldsymbol{A}^3 &= \boldsymbol{U}_1\boldsymbol{R}_1\boldsymbol{U}_1\boldsymbol{U}_2\boldsymbol{R}_2\boldsymbol{R}_1 = \boldsymbol{U}_1\boldsymbol{U}_2\boldsymbol{R}_2\boldsymbol{U}_2\boldsymbol{R}_2\boldsymbol{R}_1 = \boldsymbol{U}_1\boldsymbol{U}_2\boldsymbol{U}_3\boldsymbol{R}_3\boldsymbol{R}_2\boldsymbol{R}_1 \\
&\vdots \\
\boldsymbol{A}^k &= \boldsymbol{U}_1\boldsymbol{U}_2\cdots\boldsymbol{U}_k\boldsymbol{R}_k\boldsymbol{R}_{k-1}\cdots\boldsymbol{R}_1,
\end{aligned}$$

which is the same thing as version 1 with

$$\boldsymbol{Q}_k = \boldsymbol{U}_1\boldsymbol{U}_2\cdots\boldsymbol{U}_k.$$

To keep track of the eigenvectors, we can restate the QR algorithm as follows.

Let $\boldsymbol{\Gamma}_0 = \boldsymbol{A}$, $\boldsymbol{Q}_0 = \mathbf{I}$. For $k = 1, 2, \ldots,$ do

$$\begin{aligned}
[\boldsymbol{U}_k, \boldsymbol{R}_k] = \mathrm{qr}(\boldsymbol{\Gamma}_{k-1}) \qquad & (\text{so } \boldsymbol{\Gamma}_{k-1} = \boldsymbol{U}_k\boldsymbol{R}_k) \\
\boldsymbol{\Gamma}_k = \boldsymbol{R}_k\boldsymbol{U}_k & \\
\boldsymbol{Q}_k = \boldsymbol{Q}_{k-1}\boldsymbol{U}_k &
\end{aligned}$$

Then $\boldsymbol{\Gamma}_k \to \boldsymbol{\Lambda}$ and $\boldsymbol{Q}_k \to \boldsymbol{V}$.

## Comments on computational complexity

The methods above have been the object of intense study over the past 50-60 years, and their cost and stability is very well understood. Notice that all of the methods cost $O(N^3)$ in the general case — this is the essential cost of solving a system of linear equations.

When $N$ is small, $O(N^3)$ is OK. For example, I have a nice desktop system (two 2.93 GHz Intel Xeons with 6 cores each, 32 GB memory), and this about how long it takes MATLAB to solve $\boldsymbol{Ax} = \boldsymbol{b}$ for different $N$:

$N = 100$, 0.0007 seconds (700$\mu$s)

$N = 1000$, 0.009 seconds (9ms)

$N = 5000$, 0.550 seconds

$N = 10,000$, 3.35 seconds

$N = 20,000$, 24.53 seconds

There are many applications where $N$ is in the millions (or even billions). In these situations, solving $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ directly is infeasible. You can roughly divide problems into three categories:

**Small scale.** $N \lesssim 10^3$. Here $O(N^3)$ algorithms are OK, and exact algorithms are appropriate.

**Medium scale.** $N \sim 10^4$. Here $O(N^3)$ is not OK, $O(N^2)$ is OK. It is getting hard to even store the matrix in memory at this point.

**Large scale.** $N \gtrsim 10^5$. Here $O(N^2)$ is not OK, $O(N^3)$ is unthinkable. We need algorithms that are $O(N)$ or $O(N \log N)$, possibly at the cost of not finding the exact solution.

It is possible to beat $O(N^3)$ if the matrix you are concerned with has certain types of **structure**. In the Technical Details section below, we look at three different examples of such matrices.

In our next two lectures, we will look at two different **iterative algorithms** for finding an appropriate solution to $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ when $\boldsymbol{A}$ is sym+def. These algorithms have the nice feature that the matrix $\boldsymbol{A}$ does not need to be held in memory — all we need is a "black box" that computes $\boldsymbol{A}\boldsymbol{x}$ given $\boldsymbol{x}$ as input. This is especially nice if you have a fast implicit method for computing $\boldsymbol{A}\boldsymbol{x}$, e.g $\boldsymbol{A}$ is sparse, or is derived from some transform that has a fast algorithm (like the FFT).

# Technical Details: Some structured matrices

Below, we will look at two types of structured matrices, identity+low rank and circulant, although plenty of other types exist. Closely related to circulant matices (but actually much harder) are Toeplitz matrices — these are basically representations of convolution in matrix form, and detailed notes on them can be found here https://goo.gl/uCTcLQ. We will also encounter *banded* matrices on the homework.

## Identity + low rank

Consider a system of the form

$$(\gamma \mathbf{I} + \boldsymbol{B}\boldsymbol{B}^{\mathrm{T}})\boldsymbol{x} = \boldsymbol{b},$$

where $\gamma > 0$ is some scalar, and $\boldsymbol{B}$ is a $N \times R$ matrix with $R < N$. These types of systems are prevalent in array signal processing and machine learning. We will see that if $R \ll N$, this system can be solved in (much) faster than $O(N^3)$ time.

Note that while $\boldsymbol{B}\boldsymbol{B}^{\mathrm{T}}$ is not at all invertible (since it is rank deficient), $\gamma \mathbf{I} + \boldsymbol{B}\boldsymbol{B}^{\mathrm{T}}$ will be. To see this, set

$$\boldsymbol{z} = \boldsymbol{B}^{\mathrm{T}}\boldsymbol{x}, \qquad \boldsymbol{z} \in \mathbb{R}^R.$$

Then we can solve the system by jointly solving for $\boldsymbol{x}$ and $\boldsymbol{z}$:

$$\gamma \boldsymbol{x} + \boldsymbol{B}\boldsymbol{z} = \boldsymbol{b} \tag{5}$$

$$\boldsymbol{B}^{\mathrm{T}}\boldsymbol{x} - \boldsymbol{z} = \mathbf{0}. \tag{6}$$

Solving the first equations (5) yields

$$\boldsymbol{x} = \gamma^{-1}(\boldsymbol{b} - \boldsymbol{B}\boldsymbol{z}),$$

and then plugging this into (6) gives us

$$\begin{aligned} \gamma^{-1}\boldsymbol{B}^{\mathrm{T}}(\boldsymbol{b}-\boldsymbol{B}\boldsymbol{z})-\boldsymbol{z}&=\mathbf{0} \\ \Rightarrow \quad (\gamma\mathbf{I}+\boldsymbol{B}^{\mathrm{T}}\boldsymbol{B})\boldsymbol{z}&=\boldsymbol{B}^{\mathrm{T}}\boldsymbol{b} \end{aligned}$$

and so

$$\boldsymbol{z}=(\gamma\mathbf{I}+\boldsymbol{B}^{\mathrm{T}}\boldsymbol{B})^{-1}\boldsymbol{B}^{\mathrm{T}}\boldsymbol{b}.$$

But notice that this is an $R \times R$ system of equations.

So it takes $O(NR^2)$ to construct $\gamma\mathbf{I}+\boldsymbol{B}^{\mathrm{T}}\boldsymbol{B}$,
then $O(R^3)$ to solve for $\boldsymbol{z}$,
then $O(NR)$ to calculate $\boldsymbol{B}\boldsymbol{z}$ (and hence find $\boldsymbol{x}$).

The dominant cost in all of this is $O(NR^2)$, which is much less than $O(N^3)$ if $R \ll N$.

**Circulant systems**.

A circulant matrix has the form

$$\boldsymbol{H}=\begin{bmatrix} h_0 & h_{N-1} & h_{N-2} & \cdots & h_1 \\ h_1 & h_0 & h_{N-1} & \cdots & h_2 \\ h_2 & h_1 & h_0 & & \vdots \\ \vdots & \vdots & & \ddots & h_{N-1} \\ h_{N-1} & & & h_1 & h_0 \end{bmatrix}$$

For $\boldsymbol{H}$ symmetric, we have $h_k = h_{N-k}$ for $k = 1, \ldots, N-1$, although symmetry does not play too big a role in exploiting this structure.

Circulant matrices have two very nice properties:

- We know their eigenvectors already — they are the discrete harmonic sinusoids, the columns of the $N \times N$ Discrete Fourier Transform (DFT) matrix).
- Transforming into the eigenbasis is **fast** thanks to the FFT (which is $O(N \log N)$).

We can write

$$\boldsymbol{H} = \boldsymbol{F}\boldsymbol{\Lambda}\boldsymbol{F}^{\mathrm{H}}, \qquad F[m,n] = \frac{1}{\sqrt{N}} e^{j2\pi mn/N}$$

and

$$\boldsymbol{H}^{-1} = \boldsymbol{F}\boldsymbol{\Lambda}^{-1}\boldsymbol{F}^{\mathrm{H}},$$

so

$$\boldsymbol{H}^{-1}\boldsymbol{b} = \underbrace{\boldsymbol{F}}_{\text{FFT, } O(N\log N);} \quad \underbrace{\boldsymbol{\Lambda}^{-1}}_{\text{diagonal weighting, } O(N);} \quad \underbrace{\boldsymbol{F}^{\mathrm{H}}\boldsymbol{b}}_{\text{FFT, } O(N\log N)}$$

$\Rightarrow$ solving an $N \times N$ system of equations can be done in $O(N \log N)$ time!

This is **fast** compared to $O(N^3)$ — on my computer, I can solve a system like this in $N = 20000$ in 800$\mu$s (compare to 24 seconds for the general case).