III. Computing the Solution to Least-Squares Problems

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

Pseudo-inverse when $\boldsymbol{A}$ has full column rank	$\widehat{\boldsymbol{x}} = (\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A})^{-1}\boldsymbol{A}^{\mathrm{T}}\boldsymbol{y}$
Pseudo-inverse when $\boldsymbol{A}$ has full row rank	$\widehat{\boldsymbol{x}} = \boldsymbol{A}^{\mathrm{T}}(\boldsymbol{A}\boldsymbol{A}^{\mathrm{T}})^{-1}\boldsymbol{y}$
Tikhonov regularization	$\widehat{\boldsymbol{x}} = (\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A} + \delta\mathbf{I})^{-1}\boldsymbol{A}^{\mathrm{T}}\boldsymbol{y}$

Here are some that we did not talk about, but are easy extensions:

Constrained least-squares:

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

has solution

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

Generalized Tikhonov regularization:

$$\min_{\boldsymbol{x}} \ \|\boldsymbol{y} - \boldsymbol{A}\boldsymbol{x}\|_2^2 + \delta\|\boldsymbol{D}\boldsymbol{x}\|_2^2,$$

has solution

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

Weighted least-squares

$$\min_{\boldsymbol{x}} \ \|\boldsymbol{W}(\boldsymbol{y} - \boldsymbol{A}\boldsymbol{x})\|_2^2,$$

has solution

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

Each of the problems on the previous page involves solving a system of **symmetric positive definite** system of equations. Specifically, with a slight abuse of notation we can think of solving any of these problems as solving a system of the form

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$$

for $\boldsymbol{x}$ where $\boldsymbol{A}$ and $\boldsymbol{b}$ are known.

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 reuse our work if we have multiple right-hand sides. We have already seen one of these factorizations several times: the eigendecomposition, or more generally the singular value decomposition, which can play a central role in solving many of the problems above, as well as other least-squares problems such as the truncated SVD for stable recovery, total least-squares, and PCA. Here we will describe how to actually compute an eigendecomposition, as well as discussing several other useful matrix factorizations.

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 $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ for a given $\boldsymbol{b}$ is easy; simply take

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

This is of course very efficient computationally; we can compute $\widehat{\boldsymbol{x}}$ 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 $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ with a single matrix-vector multiply:

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

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*:

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

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 $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ 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 $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{U}$, where $\boldsymbol{L}$ is lower diagonal, and $\boldsymbol{U}$ is upper diagonal. 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 $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$, 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. See the technical details at the end of these notes for an example worked out in detail.

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 1's along the diagonal. Algorithms used to compute Cholesky factorization are similar to those that compute $\boldsymbol{LU}$ factorizations. See the technical details at the end of these notes for further detail.

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

Example.

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

$\boldsymbol{QR}$ factorization

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

$$\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{R},$$

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 $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ 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 $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ using $\boldsymbol{x} = \boldsymbol{R}^{-1}\boldsymbol{Q}^{\mathrm{T}}\boldsymbol{b}$ in $O(N^2)$ time.

Symmetric $\boldsymbol{QR}$

When $\boldsymbol{A}$ is symmetric, we can write

$$\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{T}\boldsymbol{Q}^{\mathrm{T}},$$

where $\boldsymbol{Q}$ is orthonormal, and $\boldsymbol{T}$ is symmetric and **tri-diagonal**:

$$\boldsymbol{T} = \begin{bmatrix} t_{11} & t_{12} & 0 & 0 & \cdots & 0 \\ t_{21} & t_{22} & t_{23} & 0 & \cdots & 0 \\ 0 & t_{32} & t_{33} & t_{34} & \cdots & 0 \\ \vdots & & & \ddots & & \vdots \\ 0 & \cdots & & 0 & t_{NN-1} & t_{NN} \end{bmatrix}.$$

This is a handy fact, since in general, tridiagonal matrices are easier to manipulate (invert, compute eigenvalues/eigenvectors of, etc) that general symmetric matrices.

We call this "symmetric $\boldsymbol{QR}$" since algorithms to compute this decomposition are very similar to those used to compute $\boldsymbol{A} = \boldsymbol{QR}$. See the technical details at the end of these notes for an example of such an algorithm.

SVD and eigenvalue decompositions

We are already familiar with the SVD 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 $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ 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 λ_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[1] $\boldsymbol{Q}_0$, then take

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

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

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

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

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

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 (a 3.8 GHz Intel i7 with 8 cores and 64 GB memory), and this about how long it takes MATLAB to solve $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ for different N:

$N = 100$: 0.0003 seconds (300μs)

$N = 1\,000$: 0.01 seconds (10ms)

$N = 5\,000$: 0.5 seconds

$N = 10\,000$: 2.6 seconds

$N = 50\,000$: 190 seconds (3.2 min)

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 may be 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 typically unthinkable. We need algorithms that are $O(N)$ or $O(N \log N)$, possibly at the cost of finding an exact solution.

We will start by looking at certain types of **structured** symmetric matrices. This structure allows us to solve $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ significantly faster than $O(N^3)$.

After this, we will consider **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 $\boldsymbol{A}$ involved FFTs, or is sparse).

Structured matrices

We will discuss three types of structured matrices, although plenty of other types exist. In each of these cases, the structure of the system allows us to do a solve in much better than $O(N^3)$ operations.

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 $\boldsymbol{z}$:

$$\gamma \boldsymbol{x} + \boldsymbol{B}\boldsymbol{z} = \boldsymbol{b} \tag{5}$$
$$\boldsymbol{B}^{\mathrm{T}}\boldsymbol{x} - \boldsymbol{z} = \boldsymbol{0}. \tag{6}$$

Solving the first equations (5) yields

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

and then plugging this into (6) gives us

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

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 (i.e. the columns of the $N \times N$ 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 $N = 20\,000$ in 800μs (compare to 24 seconds for the general case).

Toeplitz systems.

Toeplitz matrices, which are matrices that are constant along their diagonals, arise in many different signal processing applications, as they are fundamental in describing the action of linear time-invariant systems. For example, suppose we observe the discrete convolution of an unknown signal $\boldsymbol{x}$ of length N and a known sequence[2] $a_0, \ldots, a_{L-1}$ of length L. We can write the corresponding matrix equation as

$$\begin{bmatrix} a_0 & 0 & \cdots & & 0 \\ a_1 & a_0 & 0 & \cdots & 0 \\ a_2 & a_1 & a_0 & \cdots & 0 \\ \vdots & & & & \\ a_{N-1} & a_{N-2} & \cdots & & a_0 \\ \vdots & & & & \\ a_{L-1} & a_{L-2} & \cdots & & a_{L-N} \\ 0 & a_{L-1} & \cdots & & a_{L-N+1} \\ \vdots & & & & \\ 0 & 0 & \cdots & \cdots & a_{L-1} \end{bmatrix} \begin{bmatrix} x[0] \\ x[1] \\ \vdots \\ x[N-1] \end{bmatrix} = \begin{bmatrix} y[0] \\ y[1] \\ y[2] \\ \vdots \\ y[N-1] \\ \vdots \\ y[L-1] \\ y[L] \\ \vdots \\ y[L+N-2] \end{bmatrix}$$

If we recover $\boldsymbol{x}$ from $\boldsymbol{y}$ using least-squares, $\widehat{\boldsymbol{x}} = \boldsymbol{A}^\dagger \boldsymbol{y} = (\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A})^{-1}\boldsymbol{A}^{\mathrm{T}}\boldsymbol{y}$, then the $N \times N$ system we need to invert, $\boldsymbol{H} = \boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}$ is also Toeplitz (and is of course symmetric and non-negative definite):

$$\boldsymbol{H} = \begin{bmatrix} h_0 & h_1 & \cdots & & h_{N-1} \\ h_1 & h_0 & h_1 & \cdots & h_{N-2} \\ h_2 & h_1 & h_0 & \cdots & h_{N-3} \\ \vdots & & & & \\ h_{N-1} & \cdots & & \cdots & h_0 \end{bmatrix}.$$

[2] We are going to save some space in this section by using subscript notation to index signals that are going in a matrix; i.e. a_k instead of $a[k]$.

Here is a quick example in MATLAB:

```
>> A = toeplitz([1; randn(5,1); zeros(3,1)], [1 zeros(1,3)])

  A =

    1.0000         0         0         0
   -0.4336    1.0000         0         0
    0.3426   -0.4336    1.0000         0
    3.5784    0.3426   -0.4336    1.0000
    2.7694    3.5784    0.3426   -0.4336
   -1.3499    2.7694    3.5784    0.3426
         0   -1.3499    2.7694    3.5784
         0         0   -1.3499    2.7694
         0         0         0   -1.3499


>> H = A'*A

  H =

   23.6023    6.8156   -5.0905    1.9151
    6.8156   23.6023    6.8156   -5.0905
   -5.0905    6.8156   23.6023    6.8156
    1.9151   -5.0905    6.8156   23.6023
```

Symmetric Toeplitz systems appear frequently in linear prediction, array processing, adaptive filtering, and other areas of statistical signal processing. An $N \times N$ Toeplitz system $\boldsymbol{H}$ can be inverted in $O(N^2)$ time using the **Levinson-Durbin** algorithm. The algorithm is relatively easy to derive, and even easier to implement. The increase in efficiency it offers is significant, as the difference between $O(N^3)$, the cost of solving the system using a general linear solver, and $O(N^2)$ is enormous even for moderate N.

Technical Details: LU, Cholesky, and Symmetric QR Factorizations

Example of LU factorization

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.

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

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

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

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

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

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!), 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 above, we have

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

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 $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}_0$ for a particular $\boldsymbol{b}_0$).

Cholesky factorization

The basic idea in Cholesky factorization is that you can use elimination to find an 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 on your own!), $\boldsymbol{R}$ is again lower-triangular. Since the inverse of a lower-triangular matrix is again lower triangular (show this on your own!), we can take

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

Symmetric QR factorization

We describe one algorithm for computing a symmetric $\boldsymbol{QR}$ factorization which is very efficient and fairly easy to understand, called *Household tri-diagonalization*.

Let $\boldsymbol{A}$ be a symmetric matrix

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ \vdots & & \ddots & \vdots \\ a_{N1} & a_{N2} & \cdots & a_{NN} \end{bmatrix},$$

and set

$$\boldsymbol{x}_1 = \begin{bmatrix} 0 \\ a_{21} \\ a_{31} \\ \vdots \\ a_{N1} \end{bmatrix}, \quad \boldsymbol{y}_1 = \begin{bmatrix} 0 \\ r_1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad \text{with } r_1 = \|\boldsymbol{x}_1\|_2.$$

From $\boldsymbol{x}_1$ and $\boldsymbol{y}_1$ we create the *Householder matrix*

$$\boldsymbol{H}_1 = \mathbf{I} - 2\boldsymbol{u}_1\boldsymbol{u}_1^{\mathrm{T}}, \qquad \boldsymbol{u}_1 = \frac{\boldsymbol{x}_1 - \boldsymbol{y}_1}{\|\boldsymbol{x}_1 - \boldsymbol{y}_1\|_2}.$$

Then by direct calculation, you can check that

$$\boldsymbol{A}_2 = \boldsymbol{H}_1\boldsymbol{A}\boldsymbol{H}_1 = \begin{bmatrix} a_{11} & r_1 & 0 & \cdots & 0 \\ r_1 & \tilde{a}_{22} & \tilde{a}_{23} & \cdots & \tilde{a}_{2N} \\ 0 & \tilde{a}_{32} & \tilde{a}_{33} & \cdots & \tilde{a}_{3N} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \tilde{a}_{N2} & \tilde{a}_{N3} & \cdots & \tilde{a}_{NN} \end{bmatrix}.$$

So applying $\boldsymbol{H}_1$ on either side eliminates all but two entries in both the first row and first column of $\boldsymbol{A}$.

Notice also that the $N \times N$ matrix $\boldsymbol{H}_1$ is orthnormal, as

$$\begin{aligned}\boldsymbol{H}_1^{\mathrm{T}}\boldsymbol{H}_1 &= (\mathbf{I} - 2\boldsymbol{u}_1\boldsymbol{u}_1^{\mathrm{T}})(\mathbf{I} - 2\boldsymbol{u}_1\boldsymbol{u}_1^{\mathrm{T}}) \\ &= \mathbf{I} - 4\boldsymbol{u}_1\boldsymbol{u}_1^{\mathrm{T}} + 4\boldsymbol{u}_1\boldsymbol{u}_1^{\mathrm{T}}\boldsymbol{u}_1\boldsymbol{u}_1^{\mathrm{T}} \\ &= \mathbf{I},\end{aligned}$$

where the last equality follows since $\boldsymbol{u}_1^{\mathrm{T}}\boldsymbol{u}_1 = \|\boldsymbol{u}_1\|_2^2 = 1$.

We can continue this process on the $(N-1) \times (N-1)$ submatrix on the lower right. We take

$$\boldsymbol{x}_2 = \begin{bmatrix} 0 \\ 0 \\ \tilde{a}_{32} \\ \tilde{a}_{42} \\ \vdots \\ \tilde{a}_{N2} \end{bmatrix}, \quad \boldsymbol{y}_2 = \begin{bmatrix} 0 \\ 0 \\ r_2 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad \text{with } r_2 = \|\boldsymbol{x}_2\|_2,$$

and

$$\boldsymbol{H}_2 = \mathbf{I} - 2\boldsymbol{u}_2\boldsymbol{u}_2^{\mathrm{T}}, \qquad \boldsymbol{u}_2 = \frac{\boldsymbol{x}_2 - \boldsymbol{y}_2}{\|\boldsymbol{x}_2 - \boldsymbol{y}_2\|_2}.$$

Then

$$\boldsymbol{A}_3 = \boldsymbol{H}_2\boldsymbol{A}_2\boldsymbol{H}_2 = \begin{bmatrix} a_{11} & r_1 & 0 & 0 & \cdots & 0 \\ r_1 & \tilde{a}_{22} & r_2 & 0 & \cdots & 0 \\ 0 & r_2 & \widehat{a}_{33} & \widehat{a}_{34} & \cdots & \widehat{a}_{3N} \\ 0 & 0 & \widehat{a}_{43} & \widehat{a}_{44} & \cdots & \widehat{a}_{4N} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \widehat{a}_{N3} & \widehat{a}_{N4} & \cdots & \widehat{a}_{NN} \end{bmatrix}$$

Repeating this procedure $N-1$ times gives us a tri-diagonal matrix

$$\boldsymbol{T} = \boldsymbol{A}_N = \boldsymbol{H}_N\boldsymbol{H}_{N-1}\cdots\boldsymbol{H}_1\boldsymbol{A}\boldsymbol{H}_1\boldsymbol{H}_2\cdots\boldsymbol{H}_N \quad = \boldsymbol{Q}^{\mathrm{T}}\boldsymbol{A}\boldsymbol{Q}.$$

where

$$\boldsymbol{Q} = \boldsymbol{H}_1\boldsymbol{H}_2\cdots\boldsymbol{H}_N.$$

Note that since all of the $\boldsymbol{H}_n$ are orthonormal, so is $\boldsymbol{Q}$.

Technical Details: Solving Toeplitz systems and the Levinson-Durbin algorithm

We start by looking at how to solve $\boldsymbol{H}\boldsymbol{v} = \boldsymbol{y}$ for a very particular right-hand side. Consider

$$\begin{bmatrix} h_0 & h_1 & \cdots & & h_{N-1} \\ h_1 & h_0 & h_1 & \cdots & h_{N-2} \\ h_2 & h_1 & h_0 & \cdots & h_{N-3} \\ \vdots & & & & \\ h_{N-1} & \cdots & & \cdots & h_0 \end{bmatrix} \begin{bmatrix} v[1] \\ v[2] \\ \vdots \\ \vdots \\ v[N] \end{bmatrix} = \begin{bmatrix} h_1 \\ h_2 \\ \vdots \\ h_{N-1} \\ h_N \end{bmatrix}. \tag{7}$$

Here the first $N-1$ entries of the "observation" vector $\boldsymbol{y}$ match the last $N-1$ entries in the first column of $\boldsymbol{H}$. This system is specialized, but not at all contrived — (7) are called the **Yule-Walker equations**, and appear in many different places in statistical signal processing. Moreover, solving systems with general right-hand sides solve systems of the form (7) as an intermediate step.

The crux of the Levinson-Durbin algorithm relies upon a seemingly innocuous fact. Let $\mathbf{J}$ be the $N \times N$ *exchange matrix* (also called the *counter identity*):

$$\mathbf{J} = \begin{bmatrix} 0 & 0 & \cdots & 0 & 0 & 1 \\ 0 & 0 & \cdots & 0 & 1 & 0 \\ 0 & 0 & \cdots & 1 & 0 & 0 \\ \vdots & & & & & \vdots \\ 1 & 0 & \cdots & & & 0 \end{bmatrix}.$$

Applying $\mathbf{J}$ to a vector $\boldsymbol{x}$ reverses the entries in $\boldsymbol{x}$. For example,

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 5 \\ -1 \\ 7 \end{bmatrix} = \begin{bmatrix} 7 \\ -1 \\ 5 \end{bmatrix}.$$

It should be clear that $\mathbf{J}^{\mathrm{T}} = \mathbf{J}$ and $\mathbf{J}^2 = \mathbf{J}\mathbf{J} = \mathbf{I}$.

Applying $\mathbf{J}$ to the left of a matrix reverses all of its columns, while applying $\mathbf{J}$ to the right reverses the rows. In particular, if $\boldsymbol{H}$ is a symmetric Toeplitz matrix, then

$$\begin{aligned}
\mathbf{J}\boldsymbol{H} &= \begin{bmatrix} 0 & 0 & \cdots & 0 & 0 & 1 \\ 0 & 0 & \cdots & 0 & 1 & 0 \\ 0 & 0 & \cdots & 1 & 0 & 0 \\ \vdots & & & & & \vdots \\ 1 & 0 & \cdots & & & 0 \end{bmatrix} \begin{bmatrix} h_0 & h_1 & \cdots & & h_{N-1} \\ h_1 & h_0 & h1 & \cdots & h_{N-2} \\ h_2 & h_1 & h_0 & \cdots & h_{N-3} \\ \vdots & & & & \\ h_{N-1} & \cdots & & \cdots & h_0 \end{bmatrix} \\
&= \begin{bmatrix} h_{N-1} & h_{N-2} & \cdots & & h_0 \\ h_{N-2} & h_{N-3} & h_{N-4} & \cdots & h_1 \\ h_{N-3} & h_{N-4} & h_{N-5} & \cdots & h_2 \\ \vdots & \vdots & & & \\ h_0 & h_1 & & \cdots & h_{N-1}, \end{bmatrix}
\end{aligned}$$

and

$$\begin{aligned}
\mathbf{J}\boldsymbol{H}\mathbf{J} &= \begin{bmatrix} h_{N-1} & h_{N-2} & \cdots & & h_0 \\ h_{N-2} & h_{N-3} & h_{N-4} & \cdots & h_1 \\ h_{N-3} & h_{N-4} & h_{N-5} & \cdots & h_2 \\ \vdots & \vdots & & & \\ h_0 & h_1 & & \cdots & h_{N-1}, \end{bmatrix} \begin{bmatrix} 0 & 0 & \cdots & 0 & 0 & 1 \\ 0 & 0 & \cdots & 0 & 1 & 0 \\ 0 & 0 & \cdots & 1 & 0 & 0 \\ \vdots & & & & & \vdots \\ 1 & 0 & \cdots & & & 0 \end{bmatrix} \\
&= \begin{bmatrix} h_0 & h_1 & \cdots & & h_{N-1} \\ h_1 & h_0 & h1 & \cdots & h_{N-2} \\ h_2 & h_1 & h_0 & \cdots & h_{N-3} \\ \vdots & & & & \\ h_{N-1} & \cdots & & \cdots & h_0 \end{bmatrix} \\
&= \boldsymbol{H}.
\end{aligned}$$

So symmetric Toeplitz matrices obey the identity

$$\begin{aligned} \boldsymbol{H} &= \mathbf{J}\boldsymbol{H}\mathbf{J} \\ \Rightarrow \mathbf{J}\boldsymbol{H} &= \boldsymbol{H}\mathbf{J} \qquad (\text{since } \mathbf{J}\mathbf{J} = \mathbf{I}) \end{aligned}$$

That is, $N \times N$ symmetric Toeplitz matrices **commute** with $\mathbf{J}$.

Also note that

$$\begin{aligned} \boldsymbol{H}^{-1} &= (\mathbf{J}\boldsymbol{H}\mathbf{J})^{-1} \\ &= \mathbf{J}^{-1}\boldsymbol{H}^{-1}\mathbf{J}^{-1} \\ &= \mathbf{J}\boldsymbol{H}^{-1}\mathbf{J} \quad (\text{since } \mathbf{J}^{-1} = \mathbf{J}), \end{aligned}$$

and so $\boldsymbol{H}^{-1}$ commutes with $\mathbf{J}$ as well:

$$\boldsymbol{H}^{-1}\mathbf{J} = \mathbf{J}\boldsymbol{H}^{-1}.$$

Important note: Even though $\boldsymbol{H}^{-1}$ does commute with $\mathbf{J}$, it is not necessarily Toeplitz. Matrices which commute with $\mathbf{J}$ are called **persymmetric**. So what the calculations above are telling us is that all Toeplitz matrices are persymmetric, all inverses of Toeplitz matrices are persymmetric, but inverses of Toeplitz matrices are not (in general) Toeplitz.

Let's see how to take advantage of these properties in solving the Yule-Walker equations. Start by parititioning off the last row and column:

$$\left[\begin{array}{cccc|c} h_0 & h_1 & h_2 & \cdots & h_{N-1} \\ h_1 & h_0 & h_1 & \cdots & h_{N-2} \\ h_2 & & \ddots & & \vdots \\ \vdots & & & \ddots & h_1 \\ \hline h_{N-1} & h_{N-2} & \cdots & & h_0 \end{array}\right] \left[\begin{array}{c} v[1] \\ v[2] \\ \vdots \\ \hline v[N-1] \\ \hline v[N] \end{array}\right] = \left[\begin{array}{c} h_1 \\ h_2 \\ \vdots \\ \hline h_{N-1} \\ \hline h_N \end{array}\right]$$

which we re-write as

$$\left[\begin{array}{c|c} \boldsymbol{H}_{N-1} & \mathbf{J}\boldsymbol{h}_{N-1} \\ \hline (\mathbf{J}\boldsymbol{h}_{N-1})^{\mathrm{T}} & h_0 \end{array}\right] \left[\begin{array}{c} \boldsymbol{z} \\ \hline \beta \end{array}\right] = \left[\begin{array}{c} \boldsymbol{h}_{N-1} \\ \hline h_N \end{array}\right]$$

where

- $\boldsymbol{H}_{N-1}$ consists of the first $N-1$ rows and columns of $\boldsymbol{H}$ (this is also Toeplitz)
- $\boldsymbol{h}_{N-1} \in \mathbb{R}^{N-1}$ contains $h_1, \ldots, h_{N-1}$

Now we would like to solve for the vector $\boldsymbol{z} \in \mathbb{R}^{N-1}$ and the scalar β. We have[3]

$$\boldsymbol{H}_{N-1}\boldsymbol{z} + \beta\mathbf{J}\boldsymbol{h}_{N-1} = \boldsymbol{h}_{N-1} \tag{8}$$

$$\boldsymbol{h}_{N-1}^{\mathrm{T}}\mathbf{J}\boldsymbol{z} + \beta h_0 = h_N. \tag{9}$$

Solving the first equation yields

$$\begin{aligned} \boldsymbol{z} &= \boldsymbol{H}_{N-1}^{-1}(\boldsymbol{h}_{N-1} - \beta\mathbf{J}\boldsymbol{h}_{N-1}) \\ &= \boldsymbol{H}_{N-1}^{-1}\boldsymbol{h}_{N-1} - \beta\boldsymbol{H}_{N-1}^{-1}\mathbf{J}\boldsymbol{h}_{N-1} \\ &= \boldsymbol{H}_{N-1}^{-1}\boldsymbol{h}_{N-1} - \beta\mathbf{J}\boldsymbol{H}_{N-1}^{-1}\boldsymbol{h}_{N-1} \quad \text{(since } \boldsymbol{H}_{N-1}^{-1} \text{ commutes with } \mathbf{J}\text{)}. \end{aligned}$$

Suppose we already had the solution to the smaller system

$$\boldsymbol{v}_{N-1} = \boldsymbol{H}_{N-1}^{-1}\boldsymbol{h}_{N-1}$$

in hand. Then we could compute $\boldsymbol{z}$ using

$$\boldsymbol{z} = \boldsymbol{v}_{N-1} - \beta\mathbf{J}\boldsymbol{v}_{N-1},$$

[3]Here we use the fact that $\mathbf{J}^{\mathrm{T}} = \mathbf{J}$.

and then plugging this into (9) gives us the **scalar** equation

$$\begin{aligned} \boldsymbol{h}_{N-1}^{\mathrm{T}}\mathbf{J}(\boldsymbol{v}_{N-1} - \beta\mathbf{J}\boldsymbol{v}_{N-1}) + \beta h_0 &= h_N \\ \Rightarrow \quad \beta &= \frac{h_N - \boldsymbol{h}_{N-1}^{\mathrm{T}}\mathbf{J}\boldsymbol{v}_{N-1}}{h_0 - \boldsymbol{h}_{N-1}^{\mathrm{T}}\boldsymbol{v}_{N-1}}, \end{aligned}$$

and so we take

$$\boldsymbol{z} = \boldsymbol{v}_{N-1} - \beta\mathbf{J}\boldsymbol{v}_{N-1},$$

and set

$$\boldsymbol{v}_N = \begin{bmatrix} \boldsymbol{z} \\ \beta \end{bmatrix} = \boldsymbol{H}_N^{-1}\boldsymbol{h}_N.$$

Moral: Given the solution to

$$\boldsymbol{H}_{N-1}\boldsymbol{v}_{N-1} = \boldsymbol{h}_{N-1}$$

the solution to

$$\boldsymbol{H}_N\boldsymbol{v}_N = \boldsymbol{h}_N$$

can be computed in $O(N)$ time (a few inner products).

So to solve the $N \times N$ system of equations $\boldsymbol{H}\boldsymbol{v}_N = \boldsymbol{h}_N$, we work "from the ground up", first solving the 1×1 system

$$\boldsymbol{H}_1\boldsymbol{v}_1 = \boldsymbol{h}_1,$$

then using the solution of this to solve the 2×2 system

$$\boldsymbol{H}_2\boldsymbol{v}_2 = \boldsymbol{h}_2,$$

and then using the solution to $\boldsymbol{H}_{N-1}\boldsymbol{v}_{N-1} = \boldsymbol{h}_{N-1}$ to solve the $N \times N$ system[4]

$$\boldsymbol{H}_N \boldsymbol{v}_N = \boldsymbol{h}_N.$$

Adding together the computational costs at each stage:

$$\begin{aligned} \text{Total cost} &= (\text{some constant})(1 + 2 + \cdots + N - 1 + N) \\ &= O(N^2). \end{aligned}$$

General right-hand sides

Solving for a general right-hand side is not much harder — it just takes twice the work. (And $2N^2$ still beats N^3 every day of the week.)

To solve

$$\boldsymbol{H}\boldsymbol{x} = \boldsymbol{y}$$

we again subdivide it into sections:

$$\left[\begin{array}{c|c} \boldsymbol{H}_{N-1} & \mathbf{J}\boldsymbol{h}_{N-1} \\ \hline (\mathbf{J}\boldsymbol{h}_{N-1})^{\mathrm{T}} & h_0 \end{array}\right] \left[\begin{array}{c} \boldsymbol{w} \\ \hline \alpha \end{array}\right] = \left[\begin{array}{c} \boldsymbol{y}_{N-1} \\ \hline y_N \end{array}\right],$$

and so

$$\begin{aligned} \boldsymbol{H}_{N-1}\boldsymbol{w} + \alpha \mathbf{J}\boldsymbol{h}_{N-1} &= \boldsymbol{y}_{N-1} \\ \boldsymbol{h}_{N-1}^{\mathrm{T}} \mathbf{J}\boldsymbol{w} + \alpha h_0 &= y_N. \end{aligned}$$

Solving the first equation:

$$\boldsymbol{w} = \boldsymbol{H}_{N-1}^{-1}\boldsymbol{y}_{N-1} - \alpha \mathbf{J}\boldsymbol{H}_{N-1}^{-1}\boldsymbol{h}_{N-1}.$$

[4] By definition, $\boldsymbol{H}_N = \boldsymbol{H}$.

Now suppose we have the following solutions in hand:

$$\begin{aligned}\boldsymbol{x}_{N-1} &= \boldsymbol{H}_{N-1}^{-1}\boldsymbol{y}_{N-1}, \quad \text{and} \\ \boldsymbol{v}_{N-1} &= \boldsymbol{H}_{N-1}^{-1}\boldsymbol{h}_{N-1}.\end{aligned}$$

Then we can again quickly solve for $\boldsymbol{w}$ and α:

$$\boldsymbol{w} = \boldsymbol{x}_{N-1} - \alpha \mathbf{J}\boldsymbol{v}_{N-1},$$

with

$$\alpha = \frac{y_N - \boldsymbol{h}_{N-1}^{\mathrm{T}}\mathbf{J}\boldsymbol{x}_{N-1}}{h_0 - \boldsymbol{h}_{N-1}^{\mathrm{T}}\boldsymbol{v}_{N-1}}.$$

So given the solutions to

$$\begin{aligned}\boldsymbol{H}_{N-1}\boldsymbol{x}_{N-1} &= \boldsymbol{y}_{N-1} \\ \boldsymbol{H}_{N-1}\boldsymbol{v}_{N-1} &= \boldsymbol{h}_{N-1},\end{aligned}$$

the solution to

$$\boldsymbol{H}_N\boldsymbol{x}_N = \boldsymbol{y}_N,$$

(and also the solution to $\boldsymbol{H}_N\boldsymbol{v}_N = \boldsymbol{h}_N$) can be computed in $O(N)$ time.

Moral: Solving the $N \times N$ symmetric Toeplitz system

$$\boldsymbol{H}\boldsymbol{x} = \boldsymbol{y}$$

can be done in $O(N^2)$ time.

What we have done above is easily extended to non-symmetric Toeplitz systems as well.