CS 4220: Numerical Analysis

Beyond linear solves + using structure

David Bindel

2026-02-13

Where we are

Summarizing the story so far:

- Computation
  - Gaussian elimination with partial pivoting (GEPP) computes $PA = LU$.
  - For A symmetric and positive definite, Cholesky is $A = R^T R$.
  - There are many LU and Cholesky variants based on different blocking.
  - Given a factorization (computed in $O(n^3)$), solving a linear system involves forward and backward substitution ($O(n^2)$).
  - In Julia, the backslash operation solves linear systems given a matrix (with an algorithm based on type) or a factorization object.
- Error bounds
  - First order bound if $(A+E)\hat{x} = b + f$ then

    $$\frac{\|\hat{x} - x\|}{\|x\|} \lesssim \kappa(A) \left(\frac{\|E\|}{\|A\|} + \frac{\|f\|}{\|b\|} \right)$$
  - Algorithm is *backward stable* if rounding errors can be recast as small relative errors in the inputs ($\|E\| = O(\epsilon_{\mathrm{mach}}) \operatorname{poly}(n) \|A\|$).
  - GEPP is *usually* backward stable (but pivot growth — where \$|U| |A| — is possible)
  - Cholesky is backward stable.
- Refinement
  - Iterative refinement can "clean up" an approximate solve.
  - Approximation must be good enough relative to $\kappa(A)$.
  - With convergence, limit is accuracy of residual calculation.

We skipped lightly past the issue of how to quickly estimate the condition number efficiently, nor did we particularly dwell on alternate pivoting schemes, how to choose block sizes, or variants on the standard error bounds (e.g. using the relative condition number). These are good topics for another class. In this lecture and the next, we will instead focus on uses of LU and Cholesky beyond solving linear systems, and then discuss factorization for systems with special structure.

Uses of factorization

So far, we have mostly focused on factoring matrix as part of the process of solving one or more linear systems. However, there are many other ways that LU and Cholesky factorizations (and related factorizations) can be useful as well! Here we will list just a few.

Determinants

Numerical analysts are often wary of determinants. They sometimes lead to scaling problems, and there are many problems that *can* be solved with determinants but *should* be solved with some other tool in the interest of efficiency and numerical stability. Nonetheless, sometimes we really do need to compute a determinant, and a factorization is the right way to do it.

As we saw when we first discussed Gaussian elimination, unpermuted LU factorization $A = LU$ can be seen as applying a series of shear transformations (which are volume preserving) in order to transform A to an upper triangular form U. Geometrically, we can think of the columns of A as edges of a parallelipiped with one corner at zero. Then the action of the Gaussian elimination is to shear this shape into an axis-aligned paralleliped, where the first edge is aligned with the x axis, the second edge lies in the (x, y) plane, and so forth. We know how to compute the volume of an axis-aligned parallelipiped: we multiply the base by the height and so on. That is, the (signed) volume of the parallelipiped with edges given by U is $\det(U) = \prod_{i=1}^n u_{ii}$.

Alternately, if we feel uncertain about the geometry of determinants and parallelipipeds but are comfortable with the Laplace expansion of the determinant in terms of minors, then we can see that the determinant of a triangular matrix is the product of the diagonal elements, and so if $A = LU$ we have

$$\det(A) = \det(L)\det(U), \quad \det(L) = \prod_{i=1}^n 1 = 1, \quad \det(U) = \prod_{i=1}^n u_{ii}.$$

Given the factorization $PA = LU$, we can similarly compute

$$\det(A) = \det(P)\det(U) = (-1)^s \prod_{i=1}^n u_{ii}$$

where $\det(P)$ is the *sign* of the permutation, i.e.

$$\det(P) = (-1)^s, \quad s = \text{number of row swaps}.$$

Low-rank approximation

For a positive definite matrix A, the *pivoted Cholesky* algorithm computes

$$PAP^T = R^T R.$$

As with partial pivoting, we choose the swaps that go into P to move the biggest plausible element up at each step; but in this case, we are choosing from the diagonal elements of the Schur complement. In Julia, this can be computed by `F = Cholesky(A, RowMaximum())`.

We know that Cholesky is backward stable without pivoting. So why consider pivoting? One reason is that we can think of pivoted Cholesky as greedily constructing a sequence of approximations

$$A \approx A_{:,I} A_{II}^{-1} A_{:,I}^T.$$

That is, we select a subset of the rows and columns of A (by permuting them to the front), adding each new row/column according to where the largest error is in the remainder (the Schur complement). This is not necessarily an optimal way to select rows and columns to form a low-rank approximation, but it is often pretty good, and serves as a good starting point for other methods.

We note that Gaussian elimination with complete pivoting computes a matrix factorization $PAQ = LU$ where P and Q are row and column permutation matrices, and truncating this factorization similarly gives us a low-rank approximation to A (albeit not a foolproof one).

Testing definiteness

If A is a symmetric positive definite matrix, the Cholesky algorithm computes a factorization $A = R^T R$ where R is nonsingular. Conversely, if $A = R^T R$ where R is nonsingular, then A is positive definite (because $x^T A x = \|Rx\|^2 > 0$ for any $x \neq 0$). Therefore, the way that we usually check for positive definiteness is by running the Cholesky algorithm and seeing whether it succeeds!

Computing inertia

The so-called LDL^T factorization of a symmetric matrix A is

$$PAP^T = LDL^T$$

where P is a permutation matrix, L is unit lower triangular, and D is diagonal. The computation of this factorization follows essentially the same pattern as Cholesky or LU. Note that if we let $X = P^T L$, then we have

$$A = X^T D X,$$

i.e. A and D are *congruent* to each other, and therefore have the same inertia. This factorization is frequently the most efficient way to compute the inertia of a matrix.

Sampling multivariate Gaussians

Suppose that Z is a random vector with independent standard normal entries. Then the random vector $Y = AZ$ is a multivariate normal with mean zero and covariance

$$\operatorname{Var}(AZ) = A \operatorname{Var}(Z) A^T = AA^T.$$

Therefore, if we want to sample from a multivariate normal with mean zero and covariance K, we can do the following:

- Compute the Cholesky factorization $K = R^T R$
- Sample Z with independent standard normal entries
- Compute a sample $Y = R^T Z$

It turns out that the Cholesky factor can be used to turn many computations involving a general multivariate Gaussian into an equivalent calculation involving a vector of independent standard normals. This conversion is sometimes called a *whitening transformation*.

Faster factorizations

The standard dense Gaussian elimination and Cholesky algorithms take $O(n^3)$ time to factor an n-by-n matrix. However, we can sometimes factor special types of matrices more quickly. The most common case is matrices that are banded or sparse. We will deal with the general sparse case in a separate lecture; for now, let's consider some simpler structures.

A Hessenberg warmup

A matrix H is *upper Hessenberg* if $h_{ij} = 0$ for $i > j + 1$. That is, an upper Hessenberg matrix looks like an upper triangular matrix plus an additional subdiagonal:

$$H = \begin{bmatrix} h_{11} & h_{12} & h_{13} & \dots & h_{1,n-1} & h_{1n} \\ h_{21} & h_{22} & h_{23} & \dots & h_{2,n-1} & h_{2n} \\ & h_{32} & h_{33} & \dots & h_{2,n-1} & h_{3n} \\ & & \ddots & \ddots & \vdots & \vdots \\ & & & & h_{n,n-1} & h_{nn} \end{bmatrix}$$

What happens if we apply Gaussian elimination to an upper Hessenberg matrix? Let's first consider the case where no pivoting is needed. To "zero out" the subdiagonal elements in the first column, we only need to subtract a multiple of the first row from the second row:

$$\begin{bmatrix} h_{11} & h_{12} & h_{13} & \dots & h_{1,n-1} & h_{1n} \\ h_{21} & h_{22} & h_{23} & \dots & h_{2,n-1} & h_{2n} \\ & h_{32} & h_{33} & \dots & h_{2,n-1} & h_{3n} \\ & & \ddots & \ddots & \vdots & \vdots \\ & & & & h_{n,n-1} & h_{nn} \end{bmatrix} \mapsto \begin{bmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1,n-1} & u_{1n} \\ l_{21} & s_{22} & s_{23} & \dots & s_{2,n-1} & s_{2n} \\ & h_{32} & h_{33} & \dots & h_{2,n-1} & h_{3n} \\ & & \ddots & \ddots & \vdots & \vdots \\ & & & & h_{n,n-1} & h_{nn} \end{bmatrix}$$

where $u_{1j} = h_{1j}$, $l_{21} = h_{21}/h_{11}$, and $s_{2j} = h_{2j} - l_{21}u_{1j}$. Because we are only updating one row, the cost of this first elimination step is $O(n)$; and the Schur complement again has upper Hessenberg structure! So we have $n-1$ elimination steps, each of which costs $O(n)$ for a total cost of $O(n^2)$.

What if we need to pivot? In the first step, there are only two things that can happen: we leave the order alone, or we swap the first two rows. In either case, the upper Hessenberg structure of the problem remains the same, and we again have an $O(n^2)$ algorithm that factors $PH = LU$ where L is *unit lower bidiagonal*:

$$L = \begin{bmatrix} 1 & & & & \\ l_{21} & 1 & & & \\ & l_{32} & 1 & & \\ & & \ddots & \ddots & \\ & & & l_{n,n-1} & 1 \end{bmatrix}$$

Tricky tridiagonal

Now consider Cholesky factorization of a symmetric tridiagonal matrix

$$T = \begin{bmatrix} \alpha_1 & \beta_1 & & & \\ \beta_1 & \alpha_2 & \beta_2 & & \\ & \ddots & \ddots & \ddots & \\ & & \beta_{n-2} & \alpha_{n-1} & \beta_{n-1} \\ & & & \beta_{n-1} & \alpha_n \end{bmatrix}.$$

The first step of Cholesky factorization computes $r_{11} = \sqrt{\alpha_1}$ and $r_{12} = \beta_1/r_{11}$, and all the other elements in the first row of R are zero! The Schur complement is simply

$$S = \begin{bmatrix} \alpha_2 - r_{21}^2 & \beta_2 & & & \\ \beta_2 & \alpha_3 & \beta_3 & & \\ & \ddots & \ddots & \ddots & \\ & & \beta_{n-2} & \alpha_{n-1} & \beta_{n-1} \\ & & & \beta_{n-1} & \alpha_n \end{bmatrix};$$

this is almost the same as the trailing submatrix of T, with only the first diagonal element changed. Hence, the Cholesky factorization of can be computed with a constant cost per step (total time $O(n)$) using the following algorithm:

```
function tridiag_cholesky(T :: SymTridiagonal)
    n = size(T)[1]
    dv = copy(T.dv)  # Diagonal elements
    ev = copy(T.ev)  # Superdiagonal elements
    for j = 1:n-1
```

```
        dv[j] = sqrt(dv[j])
        ev[j] /= dv[j]
        dv[j+1] -= ev[j]^2
    end
    dv[n] = sqrt(dv[n])
    Bidiagonal(dv, ev, :U)  # Return an upper bidiagonal
end
```

One can also simply call the Julia `cholesky` routine on a `SymTridiagonal` input matrix to get this — Julia will do the right thing based on the input matrix type.