2023-02-13

1 Cholesky

So far, we have focused on the LU factorization for general nonsymmetric matrices. There is an alternate factorization for the case where A is *symmetric positive definite* (SPD), i.e.

- $A = A^T$,
- $x^T A x > 0$ for any $x \neq 0$.

For such a matrix, the *Cholesky factorization* is

$$A = LL^T \quad \text{or} \quad A = R^T R$$

where L is a lower triangular matrix with positive diagonal and R is an upper triangular matrix with positive diagonal ($R = L^T$). The Cholesky factor exists iff A is positive definite; in fact, the usual way to test numerically for positive definiteness is to attempt a Cholesky factorization and see whether the algorithm succeeds or fails. And, unlike the LU factorization, the Cholesky factorization is simply backward stable — no appeal to pivot growth factors is required.

The Cholesky algorithm looks like Gaussian elimination. As with Gaussian elimination, we figure out what goes on by block 2-by-2 factorization:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} L_{11}^T & L_{21}^T \\ 0 & L_{22}^T \end{bmatrix} = \begin{bmatrix} L_{11}L_{11}^T & L_{11}L_{21}^T \\ L_{21}L_{11}^T & L_{21}L_{21}^T + L_{22}L_{22}^T \end{bmatrix}$$

Working block-by-block, we have

$$\begin{aligned} L_{11}L_{11}^T &= A_{11} \\ L_{21} &= A_{21}L_{11}^{-T} \\ L_{22}L_{22}^T &= A_{22} - L_{21}L_{21}^T \end{aligned}$$

That is, we factor the leading block, do a solve to get the off-diagonal part, and then form and factor a Schur complement system.

Note that the Schur complement

$$A_{22} - L_{21}L_{21}^T = A_{22} - A_{21}A_{11}^{-1}A_{12}$$

is the same Schur complement that we see in Gaussian elimination with partial pivoting; and, as in Gaussian elimination, we can interpret the Schur complement as the inverse of a submatrix of A^{-1}. This is important because any principal submatrix of an SPD matrix is SPD and inverses of SPD matrices are SPD, so the Schur complements formed during Cholesky factorization remain SPD.

In terms of basic Julia operations, Cholesky factorization looks like

```
# Overwrite the lower triangular factor of A with L
for j = 1:n
  A[j,j] = sqrt(A[j,j])
  A[j+1:n,j] /= A[j,j]
  A[j+1:n,j+1:n] -= A[j+1:n,j]*A[j,j+1:n]'
end
```

This is very similar to the standard Gaussian elimination loop. The only place where we might be concerned is that we could get into trouble if we ever encountered a zero or negative diagonal element; but the fact that the Schur complements remain SPD, together with the fact that the diagonals of an SPD matrix are all positive, suffices to guarantee this will never happen.

2 Iterative refinement

If we know A and b, a reasonable way to evaluate an approximate solution $\hat{x}$ is through the residual $r = b - A\hat{x}$. The approximate solution satisfies

$$A\hat{x} = b + r,$$

so if we subtract of $Ax = b$, we have

$$\hat{x} - x = A^{-1}r.$$

We can use this to get the error estimate

$$\|\hat{x} - x\| = \|A^{-1}\|\|r\|;$$

but for a given $\hat{x}$, we also actually have a prayer of *evaluating* $\delta x = A^{-1}r$ with at least some accuracy. It's worth pausing to think how novel this situation is. Generally, we can only *bound* error terms. If I tell you "my answer is off by just about 2.5," you'll look at me much more sceptically than if I tell you "my answer is off by no more than 2.5," and reasonably so. After all, if I

knew that my answer was off by nearly 2.5, why wouldn't I add 2.5 to my original answer in order to get something closer to truth? This is exactly the idea behind *iterative refinement*:

1. Get an approximate solution $A\hat{x}_1 \approx b$.
2. Compute the residual $r = b - A\hat{x}_1$ (to good accuracy).
3. Approximately solve $A\,\delta x_1 \approx r$.
4. Get a new approximate solution $\hat{x}_2 = \hat{x}_1 + \delta x_1$; repeat as needed.

3 Multiple right hand sides

The simplest case of solving multiple problems is when the matrix is fixed, but there are several right hand sides. That is, we want to solve

$$Ax^{(k)} = b^{(k)}$$

for $k = 1, \ldots, m$. In the simple case where all the right hand sides are known in advance, we can still accomplish this by using the magic of Julia's backslash:

```
X = A\B;
```

But in some cases, we might not know the kth right hand side until we have learned the answer to the $k-1$th question. For example, suppose we wanted to run the iterative refinement process

$$x^{(k+1)} = x^{(k)} + \hat{A}^{-1}(b - Ax^{(k)})$$

that was mentioned previously. In Julia, if we had already computed the factorization

```
F = lu(A)
```

we might run the iteration

```
x = F\b
for k = 1:niter
  r = b-A*x
  x += F\r
end
```

Note that we *never* form the inverse of A, explicitly or implicitly. Rather, we apply A^{-1} to vectors through triangular solves involving the factors computed through Gaussian elimination. Using only triangular solves is good for performance (we take $O(n^2)$ time per solve after the original factorization, rather than $O(n^3)$ time); and it is good for numerical stability.

The admonition against inverses sometimes causes a certain amount of confusion, and it bears repeating: *we want to only do permutations and triangular solves* applied to vectors. Specifically, in Julia, we have

```
# Probably best
F = lu(A)
x = F\b

# Also OK
L, U, p = lu(A)
x = U\(L\b[p])

# Generally bad
x = inv(A)*b;   # Code that calls 'inv' deserves skepticism
x = U\L\b[p];   # Order of operations means we form U\L!
```