CS 4220: Numerical Analysis

Blocked LU and Cholesky

David Bindel

2026-02-09

LU, take 2

In the last lecture, we described Gaussian elimination as applying a series of Gauss transforms $M_1, \ldots, M_{n-1}$ such that

$$M_{n-1} \ldots M_1 A = U$$

where U is an upper triangular matrix. We then observed that we can summarize these operations as

$$A = LU, \quad L = M_1^{-1} \ldots M_{n-1}$$

where the L matrix are exactly the multipliers that appear during the Gaussian elimination process.

If we know that we can write $A = LU$ (absent the issue of running into a diagonal pivot), we can also figure out what L and U must be without directly thinking about manipulating a series of Gauss transformations. To do this, it is useful to take a block matrix perspective. We do this in two ways.

Take 1: Right-looking LU

Let $A \in \mathbb{R}^{n \times n}$ be written as

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & A_{22} \end{bmatrix}$$

where a_{11} is a scalar, a_{12} and a_{21} are row and column vectors of length n, and $A_{22} \in \mathbb{R}^{(n-1)\times(n-1)}$. We can similarly partition L and U. With respect to this partitioning, the equation $A = LU$ looks like

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ l_{21} & L_{22} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} \\ 0 & U_{22} \end{bmatrix}.$$

We can separate this into four equations, one for each block of A:

$$\begin{aligned} a_{11} &= u_{11} \\ a_{12} &= u_{12} \\ a_{21} &= l_{21} u_{11} \\ A_{22} &= l_{21} u_{12} + L_{22} U_{22} \end{aligned}$$

If we rearrange the equations, we have a recursive algorithm for computing the factorization

$$\begin{aligned} u_{11} &= a_{11} \\ u_{12} &= a_{12} \\ l_{21} &= a_{21}/u_{11} \\ L_{22} U_{22} &= A_{22} - l_{21} u_{12} \end{aligned}$$

The expression $A_{22} - l_{21} u_{12}$ is the *Schur complement* in the matrix — we will see later that this has meaning beyond "weird thing that shows up in Gaussian elimination." In code, we can write a function that recursively replaces A by the packed L and U factors:

```
function lu1(A)
    n = size(A)[1]
    if n != 1
        A[2:n,1] ./= A[1,1]               # Compute l21
        A[2:n,2:n] .-= A[2:n,1]*A[1,2:n]'  # Schur complement
        lu1(view(A,2:n,2:n))
    end
    A
end
```

We can equally well write this with loops rather than recursively.

If we sanity check with a randomly generated 4-by-4 matrix, we find that the L and U factors computed by `lu1` have a relative error on the order of machine epsilon, as we would hope:

```
let
    A = [0.484855   0.370397    0.528243    0.553611;
         1.0394     0.614561   -0.446556   -0.561344;
         0.831893   0.777628    0.803044    0.774805;
         1.68925   -0.0730347   0.0843504  -0.290536]
    LU = lu1(copy(A))
    L = UnitLowerTriangular(LU)
    U = UpperTriangular(LU)
    norm(L*U-A)/norm(A)
end
```

```
3.0095111196400583e-16
```

Take 1: Left-looking LU

There is nothing sacred about partitioning A the way that we did. An equally good approach is to write

$$A = \begin{bmatrix} A_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

where $A_{11} \in \mathbb{R}^{(n-1)\times(n-1)}$, a_{12} and a_{21} are a column and row vector, respectively, and a_{22} is scalar. As before, we partition L and U in the same way as A, and get four block equations

$$\begin{aligned} A_{11} &= L_{11}U_{11} \\ a_{12} &= L_{11}u_{12} \\ a_{21} &= l_{21}U_{11} \\ a_{22} &= l_{21}u_{12} + u_{22}. \end{aligned}$$

We can now rewrite the equations as a way of computing the pieces of L and U:

$$\begin{aligned} L_{11}U_{11} &= A_{11} \\ u_{12} &= L_{11}^{-1}a_{12} \\ l_{21} &= a_{21}U_{11}^{-1} \\ u_{22} &= a_{11} - l_{21}u_{12}. \end{aligned}$$

In code, we can write this as:

```
function lu2(A)
    n = size(A)[1]
    if n != 1
        A11 = view(A,1:n-1,1:n-1)
        a21 = view(A,1:n-1,n)
        a12 = view(A,n,1:n-1)
        lu2(A11)                                # Factor A11
        a21[:] = UnitLowerTriangular(A11)\a21  # Compute u21
        a12[:] = a12'/UpperTriangular(A11)     # Compute l12
        A[n,n] -= a21'*a12                      # Compute u22
    end
    A
end
```

Again, we can also write this with loops rather than recursively.

If we sanity check with a randomly generated 4-by-4 matrix, we find that the L and U factors computed by `lu2` again have a relative error on the order of machine epsilon:

```
let
    A = [0.484855   0.370397    0.528243    0.553611;
         1.0394     0.614561   -0.446556   -0.561344;
         0.831893   0.777628    0.803044    0.774805;
         1.68925   -0.0730347   0.0843504  -0.290536]
    LU = lu2(copy(A))
    L = UnitLowerTriangular(LU)
    U = UpperTriangular(LU)
    norm(L*U-A)/norm(A)
end
```

```
1.5129012086913948e-16
```

General blocking

There is nothing sacred about partitioning A so as to "pull off" a single row and column of the factorization from the beginning or end. We can equally well write

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

where A_{11} and A_{22} are square matrices. If we partition L and U in the same way, we have

$$\begin{aligned} L_{11}U_{11} &= A_{11} \\ U_{12} &= L_{11}^{-1}A_{12} \\ L_{21} &= A_{21}U_{11}^{-1} \\ L_{22}U_{22} &= A_{22} - L_{21}U_{12}. \end{aligned}$$

That is, we factor the A_{11} submatrix, do triangular solves with multiple right hand sides to compute U_{12} and L_{21}, form a Schur complement, and then recursively factor that Schur complement. If the leading submatrix fits into a low level of cache, this organization has the advantage that we can do a lot of work in the initial factorization and the formation of the Schur complement in a way that has very good memory locality and uses a lot of level 3 BLAS operations. This is what we want for good performance.

Sherman-Morrison

Manipulating block matrices is not just useful for thinking about Gaussian elimination! As an example, consider the problem of solving

$$(A + bc^T)x = f$$

for a fixed A but many possible choices of b, c, and f. We could, of course, re-factor A (at cost of $O(n^3)$) for every new b, \$c, and f — but there is a better way. We know from our discussion of matrix multiply that in general we want to avoid forming rank-one matrices explicitly; instead, it's a good idea to write

$$(bc^T)x = b(c^T x) = by, \quad y = c^T x.$$

If we explicitly think of y as an unknown and $y = c^T x$ as part of our system of equations, we have

$$\begin{bmatrix} A & b \\ c^T & -1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} f \\ 0 \end{bmatrix}.$$

Now we can do block Gaussian elimination on this linear system. First we subtract a multiple of the first block row from the second to get

$$\begin{bmatrix} A & b \\ 0 & -1 - c^T A^{-1} b \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} f \\ -c^T A^{-1} f \end{bmatrix}.$$

Now block back-substitution gives us

$$\begin{aligned} y &= \frac{c^T A^{-1} f}{1 + c^T A^{-1} b} \\ x &= A^{-1}(f - by) \end{aligned}$$

Substituting the expression for y from the first equation into the second equation gives us the Sherman-Morrison formula

$$x = A^{-1} f - \frac{A^{-1} b c^T A^{-1} f}{1 + c^T A^{-1} b}.$$

It is possible to memorize or look up this formula, but when I need it, I usually just re-derive it in terms of Gaussian elimination on a block matrix, as written here.

Cholesky factorization

When $A \in \mathbb{R}^{n \times n}$ is symmetric and positive definite, we usually avoid LU and instead use the *Cholesky factorization* $A = R^T R$ where R is upper triangular. We can derive the factorization recursively as we did for LU. Let

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{12}^T & A_{22} \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} \\ 0 & R_{22} \end{bmatrix}^T \begin{bmatrix} r_{11} & r_{12} \\ 0 & R_{22} \end{bmatrix}.$$

Because of symmetry, we can completely describe this picture with three (rather than four) block equations:

$$\begin{aligned} a_{11} &= r_{11}^2 \\ a_{12} &= r_{11} r_{12} \\ A_{22} &= r_{12}^T r_{12} + R_{22}^T R_{22}. \end{aligned}$$

As before, we rearrange these to get an algorithm for computing pieces of R:

$$\begin{aligned} r_{11} &= \sqrt{a_{11}} \\ r_{12} &= a_{12}/r_{11} \\ R_{22}^T R_{22} &= A_{22} - r_{12}^T r_{12}. \end{aligned}$$

Note that $a_{11} > 0$ by positive definiteness of A, and therefore r_{12} is also well defined. It turns out (and is left as an exercise) that the Schur complement $A_{22} - r_{12}^T r_{12}$ must also be positive definite, and so the recursion for computing the Cholesky factorization will run to completion without issue.

As with LU factorization, the Cholesky factorization can be computed in various ways associated with different blockings of the matrix. Unlike LU factorization, the Cholesky factorization never requires that we do any pivoting – a topic we turn to next.

Pivoting

In a first linear algebra class, we learn that we need row permutations to be able to solve systems with matrices like

$$A = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$$

We cannot substract a multiple of the first row from the second row in order to put a zero into the $(2,1)$ slot; if we try to apply the standard algorithm, we run into division by zero. Everything works fine if we swap the two rows and factor

$$PA = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}, \quad P = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

What happens if we replace the zero with something tiny? That is, consider now the factorization of

$$A = \begin{bmatrix} \delta & 1 \\ 1 & 1 \end{bmatrix}$$

where δ is nonzero but tiny (less than ϵ_{mach}). In this case, we don't need to pivot to compute an LU factorization in exact arithmetic:

$$\begin{bmatrix} \delta & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \delta^{-1} & 1 \end{bmatrix} \begin{bmatrix} \delta & 1 \\ 0 & 1 - \delta^{-1} \end{bmatrix}.$$

However, even if δ and δ^{-1} are exactly representable in floating point, $1 - \delta^{-1}$ will round to $-\delta^{-1}$; and

$$\begin{bmatrix} 1 & 0 \\ \delta^{-1} & 1 \end{bmatrix} \begin{bmatrix} \delta & 1 \\ 0 & -\delta^{-1} \end{bmatrix} = \begin{bmatrix} \delta & 1 \\ 1 & 0 \end{bmatrix}.$$

That is, a small relative error due to rounding the entries of U leads to an enormous relative error in A. The problem, fundamentally, is that we have allowed the entries of L and U to get absolutely enormous, much larger than the entries of A, setting ourselves up for small errors in the coefficients of those matrices to be enormous relative to A. To (mostly) fix this problem, we will turn in the next lecture to *Gaussian elimination with partial pivoting* (GEPP).