CS 542G: Pivoting in LU

Robert Bridson

September 24, 2008

1 More Variants of LU

Last time we saw a recursive divide-and-conquer version of LU alongside a traditional "right-looking" algorithm (the natural analogue of classic row reduction). There are a few other variations that have been singled out for naming in numerical linear algebra, such as the **bordering** scheme. For bordering we take the same 2×2 block partitioning:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{pmatrix}$$

but now we take A_{11} to be $(n-1) \times (n-1)$ and A_{22} to be 1×1. Algorithmically, this means after factoring A_{11} as $L_{11}U_{11}$, we can compute the last row of L and last column of U as solutions of triangular systems:

$$\begin{aligned} U_{11}^T L_{21}^T &= A_{21}^T \\ L_{11}U_{12} &= A_{12} \end{aligned}$$

and finally the last entries in L and U:

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

There is also a **left-looking** scheme which computes a column of L and of U at each step, referring just to previously computed columns (to the left), and an **up-looking** scheme which does the same with rows; they both can be derived from slightly finer block partitions.

These other schemes often enjoy a performance advantage over the right-looking scheme, due to more favourable memory traffic. However, performance for all the algorithms can be significantly improved, and largely made identical, with **block** versions of LU. For example, the block right-looking

scheme is based on a partition with A_{11} a $k \times k$ matrix for some small fixed k (usually chosen to reflect the size of cache lines and/or L1 cache, determined after some experimentation). One step of the algorithm proceeds as:

- Factor the $k \times k$ block A_{11} as $L_{11}U_{11}$ with any regular algorithm (storing L_{11} and U_{11} at least temporarily in a contiguous $k \times k$ matrices, instead of scattered across memory as a part of A; this keeps them in cache).
- Find the first k columns of L and k rows of U with block triangular solves:

$$\begin{aligned} U_{11}^T L_{21}^T &= A_{21}^T \\ L_{11}U_{12} &= A_{12} \end{aligned}$$

- Subtract from A_{22} the rank k matrix $L_{21}U_{12}$.

The last two steps can be done with optimized BLAS level 3 operations very efficiently.

As a final note, all of the algorithms we've discussed can be done more or less in-place. Since we know the diagonal of L has to all be ones, it needn't be stored explicitly, meaning the important entries of L (below the diagonal) and U (above and including the diagonal) take up exactly the same amount of space as the input A does. Moreover, once we've computed an entry in L or U, the entry of A in the same location is no longer needed and thus can be overwritten instead. This is what standard implementations do: destructively udpate A into a packed storage of L and U. Block versions of the algorithms may need some extra "elbow room" to store temporary results efficiently (such as the LU factors of the leading $k \times k$ submatrix in the block right-looking scheme) but this overhead is generally quite small.

1.1 Why?

As an aside, you may be questioning why we're spending so much time on variations of the same algorithm—that essentially amount to reordering loops without changing the final result. There are a few reasons that I should make explicit.

The first is simply to provide a path from the row reduction you're familiar with to practical algorithms like a block version of right-looking LU.

The second goal is to provide an example of how scientific computing as a process works (as opposed to simply picking out the current best recipes). Row reduction works, but by bringing in a higher level abstraction of this nuts-and-bolts algorithm—reinterpreting it as a matrix factorization—we

can get a number of advantages. This is a common theme in the course and the history of the subject, and a model for what you might do when tackling a hard problem further on in your career.

Finally, numerical linear algebra is almost always the core of scientific computing, and so it's worth the effort to get up to speed on common terminology, different ways of approaching the same basic problem (blocking is always a big deal for performance; the difference between, say, left-looking and right-looking approaches is far larger when dealing with sparse matrices.), and in general gaining familiarity with core linear algebra operations.

2 Numerical Pivoting

Our algorithms so far have a serious flaw: some perfectly well conditioned matrices don't have an LU factorization. For example:

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

has condition number $\kappa(A)$ equal to one, the best it can possibly be (it's just a permutation of the identity matrix). However, if given to any of the algorithms discussed so far, we would immediately hit a divide-by-zero in computing L_{21}, giving `inf`.

Even if the LU factorization exists in exact arithmetic, it may be numerically useless:

$$A = \begin{pmatrix} 10^{-16} & 1 \\ 1 & 0 \end{pmatrix}$$

is very nearly perfectly conditioned, but it's LU factorization is terrible:

$$A = \begin{pmatrix} 1 & \\ 10^{16} & 1 \end{pmatrix} \begin{pmatrix} 10^{-16} & 1 \\ & -10^{16} \end{pmatrix}$$

The condition number of each of these is on the order of 10^{32}, making them useless in terms of solving a linear system with A. The problem was that we had to divide by the diagonal entry 10^{-16}, which is much smaller than other entries in the matrix, leading to a massive blow-up.

The classic row reduction strategy for dealing with a zero coefficient on the diagonal is to reorder the remaining equations, picking one with a nonzero coefficient (there must be one if the matrix is non-singular). We can do the same here, which amounts to swapping the first row of A with another which has a nonzero first entry, and of course making a record of this swap to apply to the right-hand-side of a linear system we want to solve with the LU factors. We can also swap the first column of A with another

column which has a nonzero first entry; this amounts to reordering the variables in the equations. Again, we need to record such a swap to be able to undo it after solving with the resulting LU factors, and thus return the solution vector to the user in the order they were expecting.

Note that for efficiency, practical implementations don't really swap rows or columns of A with actual memory copies, but rather keep an auxiliary integer array or two with the current remapping of the rows or columns (initially set to $[0, 1, 2, 3, \ldots, n-1]$ but updated by swapping two entries when needed) that can be used to indirectly look things up as if the matrix had been reordered.

This process is called **pivoting**, or more fully **numerical pivoting** (since it depends on the numerical values of the entries). It's most clearly implemented in the context of the right-looking algorithm, which we will assume for now. In an effort to avoid dividing by small numbers (potentially causing a huge growth in the size of entries in the matrix, as seen above) pivoting determines swaps that put a large entry in the top-left position in the matrix.

Complete pivoting (or full pivoting) does this by finding the largest magnitude entry in the entire matrix and swapping its row and column with the first to get it in the $(1, 1)$ position (this entry is then called the next "pivot"). Clearly, if the matrix is nonsingular, this entry has to be nonzero and thus the right-looking factorization can proceed. It should also be clear from the right-looking formulas that U_{11} will be the largest entry in the first row of U (more generally, the largest entry in every row of U will be on the diagonal) and similarly $L_{11} = 1$ will be the largest entry in the first column of L (more generally, all entries in L are bounded by 1 in magnitude). With a little more effort it can be shown that U can't be too much larger than A in terms of norm, and most importantly that together L and U can't be too much worse conditioned than A, meaning complete pivoting is a stable algorithm that can be trusted to deliver accurate solutions to linear systems. Unfortunately, complete pivoting is usually deemed too expensive. The problem is that searching for the next pivot requires $O(n^2)$ operations, the same as the rank one update, increasing run-time by a substantial factor.

(As an aside: the fact that L is unit lower triangular with all entries bounded by one doesn't imply it's well-conditioned as I mistakenly claimed in class. You might find it an interesting exercise to find such an L that, in the $n \times n$ case, has condition number $\kappa(L) \in O(2^n)$.)

Partial pivoting is much more commonly used. Here only the first column of A is searched for the entry of largest magnitude, and the rows are appropriately swapped to bring this one to $(1, 1)$. Columns are never permuted in partial pivoting. This means L has all its entries bounded by 1, as before, but U might be worse behaved. In fact, it's possible to construct a well-conditioned matrix A where U has exponentially increasing entries, resulting in an unusably bad factorization. However, it appears that such matrices simply don't arise in real applications even though the exact reasons for this remain mysterious.

Decades of experience suggest that LU with partial pivoting is reliable for all real-world matrices. Since the search for the pivot only takes $O(n)$ time per step, it's much more efficient than complete pivoting, and has the added advantage that other variations of LU such as the left-looking algorithm can be used (complete pivoting requires a right-looking approach).

3 LAPACK

This leads us to the next significant standard scientific computing library, LAPACK (short for Linear Algebra Package), with code and documentation available at `www.netlib.org/lapack`. Unlike the BLAS, this is more than an API but a full library; there basically is only one implementation. It provides efficient and robust (very well tested!) routines for solving linear systems with LU factorization, along with many other useful operations such as solving eigenvalue problems.

It derives its efficiency mostly by implementing algorithms in a block fashion, as we saw earlier, and using calls to BLAS (wherever possible at level 3) to exploit all the machine-specific optimization that has been done for the BLAS. This means LAPACK doesn't need tuning or reoptimizing on each platform: one code base works well for everyone.

LAPACK, in the case of LU factorization, also includes the capability of estimating the condition number of A after factorization. Recall $\kappa(A) = \|A\|\|A^{-1}\|$, and that the matrix 1-norm $\|A\|_1$, for example, is easy enough to compute; it's the norm of the inverse $\|A^{-1}\|$ that's tricky. However, with the definition

$$\|A^{-1}\|_1 = \max_{\|x\|_1=1} \|A^{-1}x\|_1$$

in mind, it turns out to be feasible to relatively efficiently find a unit-norm vector x which usually comes close to maximizing this quantity, making use of solves with L and U to apply A^{-1} to candidate vectors. (Look up N. Higham's paper "FORTRAN Codes for Estimating the One-Norm of a Real or Complex Matrix, with Applications to Condition Estimation", ACM TOMS, 14(4), 1988 for more on how LAPACK does this.)

As with the BLAS, if you need to do an operation covered by LAPACK, it's almost never a good idea to write your own version; despite the FORTRAN calling conventions and the hassle of linking against the right libraries, it will probably be a lot easier than getting your own robust and reasonably fast version working.

4 Specializing LU

Finally, we'll take a peek at some special classes of matrices A for which the LU factorization is simplified and possibly sped up significantly.

The most common case is when A is synmmetric: $A = A^T$. This includes the RBF system, for example, and maybe even the majority of matrices that arise in the real world. The goal is that somehow LU can be modified to exploit that symmetry, speeding things up by a factor of two. Specializing further, the important class of **symmetric positive definite** matrices (abbreviated SPD) in particular leads to an especially efficient and elegant version of LU.

A matrix being positive definite is usually defined as the following property:

$$\begin{aligned} x^T Ax &\geq 0 \qquad \text{for any vector } x \\ x^T Ax &= 0 \qquad \text{only if } x = 0 \end{aligned}$$

For a symmetric matrix, this is also equivalent to requiring all the eigenvalues are positive (recall that the eigenvalues of a symmetric matrix have to be real, but in general could be of any sign).

Similarly you can define **negative definite** with the inequality the other way: A is negative definite if and only if $-A$ is positive definite.

Positive or negative **semidefinite** matrices relax the second condition, allowing $x^T Ax = 0$ for nonzero x.

Finally, if a matrix isn't definite or semidefinite, it is termed **indefinite**: for some vectors x the product $x^T Ax > 0$ but for others $x^T Ax < 0$. It's relatively easy to determine that the symmetric matrices we've seen so far (for RBFs and the 2×2 examples seen above) are indefinite; this makes them harder to solve, and so next we'll concentrate instead on the easier symmetric positive definite case.