Intro to Numerical Linear Algebra for AMSC 460
Lecture 3

Prof. Jacob Bedrossian
University of Maryland, College Park

These notes will supplement the lectures on numerical linear algebra. NLA is one of the most important, if not the most important, branches of numerical analysis. For many scientific computing calculations, most of the compute time is spent doing linear algebra, and so doing NLA effectively is crucial to useful code. Probably the best reference I've ever seen to introduce the topic of NLA is the book Trefethen and Bau, but its too advanced to follow directly in AMSC 460. These notes are a distillation of Bindle+Goodman's notes on scientific computing and Trefethen and Bau's book.

1 LU factorization with partial pivoting

The algorithm we discussed before is not really usable. For example, if we attempt to solve the simple linear system:

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix},$$

our algorithm will fail immediately when it tries to divide by zero. This system is easily solvable by hand, there is no pathology at all. The situation is in some ways even worse if one considers the very similar (also quite benign) matrix (this example is from Trefethen and Bau),

$$A = \begin{pmatrix} 10^{-20} & 1 \\ 1 & 1 \end{pmatrix}$$

Our algorithm will produce the LU factorization

$$L = \begin{pmatrix} 1 & 0 \\ 10^{20} & 1 \end{pmatrix}, \qquad U = \begin{pmatrix} 10^{-20} & 1 \\ 0 & 1 - 10^{20} \end{pmatrix}.$$

This is indeed an LU factorization of A. However in floating point arithmetic, the $1 - 10^{20}$ would get rounded to -10^{20}, since the 1 is more than 16 orders of magnitude smaller than 10^{20}. Hence, our numerical algorithm would return two approximations for L and U:

$$\tilde{L} = \begin{pmatrix} 1 & 0 \\ 10^{20} & 1 \end{pmatrix}, \qquad \tilde{U} = \begin{pmatrix} 10^{-20} & 1 \\ 0 & -10^{20} \end{pmatrix}.$$

However, that 1 was important, since

$$\tilde{L}\tilde{U} = \begin{pmatrix} 10^{-20} & 1 \\ 1 & 0 \end{pmatrix}$$

which is not even remotely close to A.

We learned how to resolve this issue when we first learned Gaussian elimination in introductory linear algebra. We should choose more wisely which rows we use to do the eliminating. The easiest and standard way of doing this is called *partial pivoting*. The idea is that instead of blindly trying to eliminate with whatever happens to be in the diagonal at the $k-th$ step, we will judiciously choose a row we want to use to eliminate all the others simply by the intuition that we want to divide by big numbers rather than little numbers. Before writing pseudo-code, lets see what this could mean in action: suppose we want to factorize this matrix:

$$A = \begin{pmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{pmatrix}. \tag{1}$$

At the start of the algorithm, we write

$$L = I, \quad U = A. \tag{2}$$

We like the idea of dividing by large numbers however, so we are going to swap the first and the third rows:

$$U = P_1 A = \begin{pmatrix} 8 & 7 & 9 & 5 \\ 4 & 3 & 3 & 1 \\ 2 & 1 & 1 & 0 \\ 6 & 7 & 9 & 8 \end{pmatrix},$$

where

$$P_1 = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Now, we want to perform a row elimination step, so just like in the previous algorithm we set the first column of L:

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 \\ \frac{1}{2} & 1 & 0 & 0 \\ \frac{1}{4} & 0 & 1 & 0 \\ \frac{3}{4} & 0 & 0 & 1 \end{pmatrix},$$
$$U = \begin{pmatrix} 8 & 7 & 9 & 5 \\ 0 & -\frac{1}{2} & -\frac{3}{2} & -\frac{3}{2} \\ 0 & -\frac{3}{4} & -\frac{5}{4} & -\frac{5}{4} \\ 0 & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \end{pmatrix}.$$

At this step, we want to use the 7/4 to eliminate the rest of the subdiagonal in the second column, so this means we have to again permute the rows. So we want to set

$$U = P_2 \begin{pmatrix} 8 & 7 & 9 & 5 \\ 0 & -\frac{1}{2} & -\frac{3}{2} & -\frac{3}{2} \\ 0 & -\frac{3}{4} & -\frac{5}{4} & -\frac{5}{4} \\ 0 & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \end{pmatrix},$$
$$P_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

and hence after the second permutation, we have

$$U = \begin{pmatrix} 8 & 7 & 9 & 5 \\ 0 & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \\ 0 & -\frac{3}{4} & -\frac{5}{4} & -\frac{5}{4} \\ 0 & -\frac{1}{2} & -\frac{3}{2} & -\frac{3}{2} \end{pmatrix}.$$

The extra tricky part is to figure out what to do with L. If you think about it for a while, since we just swapped the rows of U, it makes sense to swap the rows of L in the columns that we have already set. This is because L is a record of the row-elimination operations we did. If we swap the corresponding rows, that swaps the order of the row eliminations, but it doesn't really change anything else about them. The row operations are still row operations associated with a lower triangular matrix. Hence, it makes sense if we could just make the following our new L:

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 \\ \frac{3}{4} & 1 & 0 & 0 \\ \frac{1}{4} & 0 & 1 & 0 \\ \frac{1}{2} & 0 & 0 & 1 \end{pmatrix},$$

and then continue on with the algorithm as if nothing happened and set the new column of L to do the corresponding row elimination step to U and update L and U accordingly:

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 \\ \frac{3}{4} & 1 & 0 & 0 \\ \frac{1}{4} & -\frac{3}{7} & 1 & 0 \\ \frac{1}{2} & -\frac{2}{7} & 0 & 1 \end{pmatrix}, \qquad U = \begin{pmatrix} 8 & 7 & 9 & 5 \\ 0 & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \\ 0 & 0 & -\frac{2}{7} & -\frac{4}{7} \\ 0 & 0 & -\frac{6}{7} & -\frac{2}{7} \end{pmatrix}$$

This actually will work (see below). Then, we are going to apply the last permutation to swap the last two rows:

$$P_3 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix},$$

and apply P_3 to U to update U as

$$U = \begin{pmatrix} 8 & 7 & 9 & 5 \\ 0 & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \\ 0 & 0 & -\frac{6}{7} & -\frac{2}{7} \\ 0 & 0 & -\frac{2}{7} & -\frac{4}{7} \end{pmatrix},$$

and we would update L accordingly like last time, permuting only the row operations that were already computed:

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 \\ \frac{3}{4} & 1 & 0 & 0 \\ \frac{1}{2} & -\frac{2}{7} & 1 & 0 \\ \frac{1}{4} & -\frac{3}{7} & 0 & 1 \end{pmatrix}$$

Then we apply the final elimination and end up with

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 \\ \frac{3}{4} & 1 & 0 & 0 \\ \frac{1}{2} & -\frac{2}{7} & 1 & 0 \\ \frac{1}{4} & -\frac{3}{7} & \frac{1}{3} & 1 \end{pmatrix}, \qquad U = \begin{pmatrix} 8 & 7 & 9 & 5 \\ 0 & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \\ 0 & 0 & -\frac{6}{7} & -\frac{2}{7} \\ 0 & 0 & 0 & \frac{2}{3} \end{pmatrix}.$$

Did we succeed in factorizing A? Nope, not at all! However, what we *did* succeed in factorizing is:

$$P_3P_2P_1A = LU. \tag{3}$$

Indeed, denoting $P = P_3P_2P_1$, we have computed the LU factorization of the following matrix:

$$PA = \begin{pmatrix} 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \\ 4 & 3 & 3 & 1 \\ 2 & 1 & 1 & 0 \end{pmatrix} = LU.$$

It might seem a bit magical that it worked out like that, and verifying that the above algorithm will always give exactly this (a row permutation matrix P and an LU factorization $PA = LU$) is not very obvious. Proving it carefully is pretty cumbersome and is more than a little confusing, so I will skip this and we will be settled with the intuition behind the algorithm. Specifically, the main unclear step to the algorithm is how we are supposed to be permuting L as we go, and this is kind of the crux of why things work.

If we believe that the above algorithm works, then we can use it very effectively. If we want to solve $Ax = b$, then we use the above algorithm to compute $P, L,$and U and solve instead the problem

$$LUx = Pb, \tag{4}$$

which is done by first solving $Ly = Pb$ and then $Ux = y$ with the forward and back substitution algorithms. This will be our plan.

In practice there are several ways one can represent P. One way (not a great way) is to simply represent it directly as a matrix of 0's and 1's and each step do a matrix multiply to update it. However, computing the associated matrix multiplications by brute force will actually be slower than the *entire factorization process* (and you should know how to show that!). A smarter and very simple way is to store the list of permutations as pairs in an ordered list, so that the sequence of row permutations can be recovered to apply to b whenever necessary. This only requires storing $2n$ integers, requires essentially no calculation, and computing Pb in the end would only require $O(n)$ operations (essentially free compared to the rest of the computations being done). A similar (and arguably even smarter) method is to store the final re-arrangement of rows, that is a list of n integers which records how to map the original rows to the final rows. Each step of the algorithm this could be updated as one continues, without having to remember the exact sequence of permutations being done. My pseudo-code will do the second option, as this is a reasonable balance between being very easy and still efficient.

Algorithm 1 (*LU* decomposition with partial pivoting – intermediate level *LU* factorization)**.** I will use #'s to denote inline comments. This is pseudo-code, not quite actual computer code.

```
U=A, L=I, P={}                                          #P will be an ordered list of pairs
for k=1 to n-1:
    Find p, k <= p <=n such that |U[p,k]| >= |U[q,k]| for all k <= q <= n
    Swap the rows U[p,:] and U[k,:]
    Swap the first k-1 elements of L[p,:] with the first k-1 elements of L[k,:]
    Add (p,k) to the list P.
    for j=k+1 to n:                                  #Now we do regular elimination step
        L[j,k] = U[j,k]/U[k,k]
        for i=k to n:
            U[j,i] = U[j,i] - L[j,k]*U[k,i]
```

Note that to compute Pb with the above representation of the permutations we can just do:

```
for (k,p) in P:
    swap b[k] and b[p]
```

Hence, this method requires $2n$ integers stored[1] and each application requires n floating point swaps...These are not really “floating point operations” per se, but it can be useful to count operations like that if you are doing a lot of them as they can take a lot of time as well[2] Thankfully, in our context, this $O(n)$ operation will be totally irrelevant to the $\frac{2}{3}n^3 + O(n^2)$ cost of our factorization or the $O(n^2)$ cost of our backwards and forwards substitution solves.

[1]In certain high performance situations it is important to be very miserly with memory allocations (as well as taking care about *precisely when* and *precisely how* such memory is allocated – hence the interest in using very explicit languages such as C/C++ in scientific computing.

[2]If you learn parallel algorithms you will find that in some situations, the computer wastes more time moving stuff around and communicating between different processors than it actually does doing flops.