Lecture 7 Notes

1 Introduction

In the last lecture, we talked about how to solve a system of equations using a process called Gaussian elimination. This method "zeroed out" the coefficients of many variables in our equations, leaving us with a new system that was much easier to solve. For instance, we started with the system

$$\begin{cases} 2x_1 + x_2 + x_3 &= 1, \\ 4x_1 + 3x_2 + 3x_3 &= 1, \\ 8x_1 + 7x_2 + 9x_3 &= -1. \end{cases}$$

We then recombined these equations to get the new system

$$\begin{cases} 2x_1 + x_2 + x_3 &= 1, \\ 0x_1 + x_2 + x_3 &= -1, \\ 0x_1 + 0x_2 + 2x_3 &= -2. \end{cases}$$

We can summarize the steps required to get this system by

$$\begin{aligned} -2 \cdot R_1 + R_2, \\ -4 \cdot R_1 + R_3, \\ -3 \cdot R_2 + R_3. \end{aligned}$$

(The first line, for example, means "multiply the first row by -2 and add it to the second row.")

We went through this work because our system is now in an upper triangular form. That is, all the coefficients below the main diagonal are zero, so the only nonzero coefficients are in a triangle in the upper right. We can easily solve an upper triangular system by solving the last equation for the last variable, then plugging this answer into the second to last equation and solving for the second to last variable, etc. Since we solve for the variables in reverse order, this is called back substitution. It would be similarly easy to solve a lower triangular system (where all the coefficients above the diagonal were zero) by solving everything in the opposite order. Such a process is called forward substitution.

2 Speed of Back/Forward Substitution

Just like with Gaussian elimination, we would like to estimate how much time it takes to solve these triangular systems. To do so, we need a rough estimate of how many flops are needed for the entire algorithm. As before, we will let N be the number of variables/equations in our system. (Also as before, we will be very cavalier with our estimates.)

Notice that each equation takes a different amount of work. The first equation we solve takes only one step (we just divide two numbers), but the last equation we solve takes much more time. To simplify our estimate, we will make a worst case estimate by assuming that every equation takes as long as the last one. How many steps does this last equation take? We have to substitute each of our $N-1$ other variables into the equation, multiply them each by a coefficient and subtract them over to the right-hand side. Almost by definition, that takes $N-1$ flops. We then have to divide both sides by the coefficient of our unknown variable, which takes one more flop. That means that (as a worst case estimate) solving for each variable takes N flops. Since there are N variables in total, back/forward substitution takes approximately N^2 flops. Again, we say that back/forward substitution is $\mathcal{O}(N^2)$ to indicate that we are ignoring constant factors or other smaller terms in our estimate.

As before, we can use this to estimate how long our algorithm takes to run. If t is the time it takes our computer to execute one flop and T is the total runtime for the algorithm, then $T \approx tN^2$. As an example, if $t = 0.000001$ seconds (which is not an unreasonable approximation) and $N = 100$, then it would take $T = 0.000001 \cdot 100^2 = 0.01$ seconds to run the whole program. Notice that this is 100 times faster than Gaussian elimination (from the last lecture). Similarly, if we wanted to solve a system that was ten times larger, so $N = 1000$, then we would need $T = 0.000001 \cdot 1000^2 = 1$ second. If we wanted to solve a problem that was ten times larger again, then we would need $T = 0.000001 \cdot 10000^2 = 100$ seconds, or a little less than 2 minutes. Compare this to Gaussian elimination, which took almost 12 days for the same size problem. The fact that N only has a power of 2 instead of 3 makes an enormous difference when we try to solve large problems.

3 LU Decomposition

We now know two ways to solve a system of equations. In general, we can use Gaussian elimination followed by back substitution, which takes $\mathcal{O}(N^3)$ flops[1]. If the system is triangular, then we get to skip the Gaussian elimination process and only use forward or back substitution (depending on if the system is lower or upper triangular), which only takes $\mathcal{O}(N^2)$ flops.

As we have seen, $\mathcal{O}(N^3)$ processes take far too much time for even moderately sized systems. It would be much more convenient if we could always use $\mathcal{O}(N^2)$ algorithms. In particular, it would be very convenient if we only had to solve triangular systems. Of course, most systems of equations are not triangular, but it turns out that we can often write a system as the combination of two triangular systems.

In particular, it turns out that we can always[2] rewrite a matrix A as the product of two triangular matrices L and U. That is, we can always write $A = LU$, where L is an $n \times n$ lower triangular matrix (and even has ones on the diagonal) and U is an $n \times n$ upper triangular matrix.

Suppose, for the moment, that someone already told you the correct L and U. That is, you are trying to solve $A\mathbf{x} = \mathbf{b}$ and you know that $A = LU$. We can rewrite our system as $LU\mathbf{x} = \mathbf{b}$. It might not seem like this is much of an improvement, but it turns out that this makes our system much easier to solve. The trick is to do it in two steps. First, we will replace $U\mathbf{x}$ with a vector $\mathbf{y}$. (This makes sense from a dimensional standpoint - U is an $n \times n$ matrix and $\mathbf{x}$ is an $n \times 1$ vector, so $U\mathbf{x}$ is still an $n \times 1$ vector.) We now have the system $L\mathbf{y} = \mathbf{b}$, but that is easy (and fast) to solve with forward substitution. Once we find $\mathbf{y}$, we can solve the equation $U\mathbf{x} = \mathbf{y}$ for $\mathbf{x}$. Once again, this system is triangular, so it is easy (and fast) to solve using back substitution. In particular, once we know L and U, solving for $\mathbf{x}$ only takes $\mathcal{O}(N^2)$ steps[3].

The real question, of course, is how we find L and U. Through a somewhat lucky coincidence, it turns out that we can find these matrices through Gaussian elimination. We will go through an example by hand and then turn to Matlab.

---

[1] You might think that we should say that it takes $\mathcal{O}(N^3) + \mathcal{O}(N^2)$, but for large N, the N^3 is so much larger than N^2 that the latter term doesn't even matter. We are also sweeping the technical definition of $\mathcal{O}$ under the rug, but if you know the definition then it is easy to show that $\mathcal{O}(N^3) + \mathcal{O}(N^2)$ is exactly the same as $\mathcal{O}(N^2)$.

[2] Actually, we can only do this *almost* always, but we will soon fix the method so that it always works.

[3] Again, you might think this should be $\mathcal{O}(2N^2)$, but when N is large the factor of 2 doesn't matter.

Remember our 3×3 system from the introduction. We were trying to solve

$$A\mathbf{x} = \begin{pmatrix} 2 & 1 & 1 \\ 4 & 3 & 3 \\ 8 & 7 & 9 \end{pmatrix} \mathbf{x} = \begin{pmatrix} 1 \\ 1 \\ -1 \end{pmatrix} = \mathbf{b}.$$

After performing the row operations

$$\begin{aligned} -2 \cdot R_1 + R_2, \\ -4 \cdot R_1 + R_3, \\ -3 \cdot R_2 + R_3, \end{aligned}$$

we obtained the new system

$$\begin{pmatrix} 2 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \mathbf{x} = \begin{pmatrix} 1 \\ -1 \\ -2 \end{pmatrix}.$$

This new system is upper triangular, and we will use the resulting matrix as U. That is,

$$U = \begin{pmatrix} 2 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix}.$$

The matrix L is somewhat more complicated, but we can create it by looking at the row operations we employed. L is always of the form

$$L = \begin{pmatrix} 1 & 0 & 0 \\ \ell_{21} & 1 & 0 \\ \ell_{31} & \ell_{32} & 1 \end{pmatrix},$$

where the entries ℓ_{ij} are numbers that we have to determine. It turns out that these entries are just the coefficients we used in our row operations with the signs reversed. For instance, we used the row operation $-2 \cdot R_1 + R_2$ to zero out the 2nd row, 1st column of A, so the entry $\ell_{21} = 2$ (note that the sign has flipped). Likewise, we used the row operation $-4 \cdot R_1 + R_3$ to change the 3rd row, 1st column of A, so $\ell_{31} = 4$. Finally, we used the row operation $-3 \cdot R_2 + R_3$ to change the 3rd row, 2nd column of A, so $\ell_{32} = 3$. This means that

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

It is easy to check (and you should do so) that $LU = A$, as desired.

Actually, Matlab doesn't quite use this method. The reason is that it is not always possible and can occasionally produce very large rounding errors. Fortunately, it turns out that we can always reorder our equations to avoid these problems. Matlab always chooses the optimal order for these equations, then performs the process we just described. We will not worry about how to find this order and just let Matlab do it for us.

It is often convenient to encode this change of order in a matrix. We will call a matrix a permutation matrix if it is the identity matrix with some of the rows reordered. For example,

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

is a permutation matrix because it is the 3×3 identity matrix with the last row moved to the top. If you multiply a permutation matrix by another matrix or vector, it just reorders the rows of the matrix/vector. For instance,

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

That is, PA is just A with the last row moved to the top. What Matlab does is reorder the rows of A with some permutation matrix P, then find the L and U corresponding to the resulting matrix PA. That is, it finds a lower triangular matrix (with ones on the diagonal) L, an upper triangular matrix U and a permutation matrix P such that $LU = PA$.

We can use the matrices L, U and P to solve a system almost as easily as we can without the P. If we know that $A\mathbf{x} = \mathbf{b}$, then we first multiply both sides by P to get $PA\mathbf{x} = P\mathbf{b}$. Since $LU = PA$, we can replace this equation with $LU\mathbf{x} = P\mathbf{b}$. As before, we now set $U\mathbf{x} = \mathbf{y}$, which leaves us with the equation $L\mathbf{y} = P\mathbf{b}$. This is easy to solve with forward substitution. Once we find $\mathbf{y}$, we can then solve $U\mathbf{x} = \mathbf{y}$ quickly using back substitution.

To see how to do this in code, we will now switch to the accompanying `.m` file.