Lecture 6 Notes

1 Dot Product

We have already seen how to add/subtract two vectors and how to multiply a vector by a single number (i.e., by a scalar). However, we have not talked about how to multiply two vectors together. For instance, if

$$\mathbf{x} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \text{ and } \mathbf{y} = \begin{pmatrix} -2 \\ 1 \\ 5 \end{pmatrix},$$

we have not defined $\mathbf{xy}$. You might expect that we would define this as

$$\mathbf{xy} = \begin{pmatrix} (1)(-2) \\ (2)(1) \\ (3)(5) \end{pmatrix} = \begin{pmatrix} -2 \\ 2 \\ 15 \end{pmatrix},$$

(and indeed, you can do exactly that in Matlab with the command `x.*y`), but we do not do so. The mathematical notion for the product of two vectors (or at least one possible notion) is called the *dot product* or *inner product* of two vectors. Instead of multiplying two vectors to produce another vector, the dot product of two vectors gives a single number. In particular, the dot product of two vectors is the sum of the products of their entries. For example,

$$\mathbf{x} \cdot \mathbf{y} = (1)(-2) + (2)(1) + (3)(5) = 15.$$

In general, if $\mathbf{x}$ and $\mathbf{y}$ are vectors with n entries, then we define

$$\mathbf{x} \cdot \mathbf{y} = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n.$$

Notice that this definition only makes sense if the two vectors have the same length. Mathematically speaking, it really only makes sense to dot two column vectors together, but we will follow Matlab's convention and not worry about whether vectors are columns or rows. This means that you can take the dot product of any two vectors (both columns, both rows, or one column and one row) as long as they have the same length. The Matlab command for this is `dot(x,y)`.

2 Matrix Multiplication

We have also already seen how to add/subtract two matrices and how to multiply a vector by a single number (i.e., by a scalar). We also briefly discussed two different ways to multiply matrices together. If the matrices have exactly the same size (that is, the same number of rows and the same number of columns) then we can do elementwise multiplication. For example, if

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \text{ and } B = \begin{pmatrix} -2 & 1 \\ 0 & 3 \end{pmatrix},$$

then the elementwise product of A and B would be

$$A. * B = \begin{pmatrix} (1)(-2) & (2)(1) \\ (3)(0) & (4)(3) \end{pmatrix} = \begin{pmatrix} -2 & 2 \\ 0 & 12 \end{pmatrix}.$$

The command for this in Matlab is `A.*B`. Despite being the obvious choice, elementwise multiplication is usually not what we want in numerical calculations. Instead, we usually define matrix multiplication as follows:

If A and B are $n \times m$ and $m \times k$ matrices (notice that the number of columns of A has to be the same as the number of rows of B), then we define the product AB as an $n \times k$ matrix where every entry is the dot product of a row of A with a column of B. For example,

$$AB = \begin{pmatrix} (1)(-2) + (2)(0) & (1)(1) + (2)(3) \\ (3)(-2) + (4)(0) & (3)(1) + (4)(3) \end{pmatrix} = \begin{pmatrix} -2 & 7 \\ -6 & 15 \end{pmatrix}.$$

A is a 2×2 matrix and B is a 2×2 matrix, so the number of columns in A is the same as the number of rows in B, which means we are allowed to multiply these two matrices. The result is a 2×2 matrix (because A has two rows and B has two columns). The entry in the first row, first column of AB is the dot product of the first row of A with the first column of B. Likewise, the entry in the first row, second column of AB is the dot product of the first row of A with the second column of B. Similarly, the entry in the second row, first column of AB is the dot product of the second row of A with the first column of B. Finally, the entry in the second row, second column of AB is the dot product of the second row of A with the second column of B.

Notice that this definition only makes sense if the number of columns of A and the number of rows of B are the same, because that ensures that we will always take the dot product of vectors with the same length. We could not multiply a 2×2 by a 3×2, for instance, because we cannot dot a vector of length 2 with a vector of length 3.

As another example, let

$$C = \begin{pmatrix} 1 & -3 & 0 \\ 2 & 1 & 4 \end{pmatrix} \text{ and } D = \begin{pmatrix} -2 & 1 & 3 \\ 4 & 2 & 0 \\ 2 & -2 & 1 \end{pmatrix}.$$

C is a 2×3 matrix and D is a 3×3 matrix, so it makes sense to multiply CD and the result will be a 2×3. (Notice that it does not make sense to multiply DC.) We get

$$\begin{aligned} CD &= \begin{pmatrix} (1)(-2)+(-3)(4)+(0)(2) & (1)(1)+(-3)(2)+(0)(-2) & (1)(3)+(-3)(0)+(0)(1) \\ (2)(-2)+(1)(4)+(4)(2) & (2)(1)+(1)(2)+(4)(-2) & (2)(3)+(1)(0)+(4)(1) \end{pmatrix} \\ &= \begin{pmatrix} -14 & -5 & 3 \\ 8 & -4 & 10 \end{pmatrix}. \end{aligned}$$

Why did we introduce such a convoluted definition? As we will see soon, it gives us an convenient way to write down systems of equations.

3 Systems of Equations

One of the most important things we will learn in this class is how to solve systems of linear equations. It turns out that such systems arise when solving a wide variety of applied problems, even if the original problem does not appear linear. What do we mean by a system of linear equations? You should already be familiar with the concept from a basic algebra class. For instance:

$$\begin{cases} 2x + y &= 5, \\ -6x + y &= -3, \end{cases}$$

or

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

(If we only have two equations, we will often call the variables x and y. If there are three equations, we will occasionally call the variables x, y and z. For anything else, we will use a single letter with different subscripts, like in the second example.) For starters, let's focus on the first set of equations. We have two equations and two variables, so we refer to this as a 2×2 system (read: two by two). You have probably seen several ways to solve this system.

As an example, we could solve the first equation for y, then substitute this into the second equation. We would get $y = 5 - 2x$, which means that

$$-6x + (5 - 2x) = -3, \text{ so } -8x = -8, \text{ so } x = 1.$$

Plugging this back into $y = 5 - 2x$, we get $y = 3$. This method is called *substitution*. It rapidly becomes messy when we try to generalize this method to larger systems, so we will only use substitution for systems in certain special forms.

You have probably seen another approach, where we try to cancel out one of the variables from one equation. For example, if we multiplied the first equation and added it to the second, we would cancel out the x's. We obtain

$$\begin{cases} 2x + y &= 5, \\ 0x + 4y &= 12. \end{cases}$$

This process is called *elimination*. Notice that we have not solved the system yet, but we made it much easier to solve by substitution. The second equation is now very easy to solve for y, and we find that $y = 3$. Now that we have this value, we can substitute it into the first equation to find $x = 1$. The system we obtained from elimination is called *upper triangular* because the nonzero coefficients form a triangle in the upper right (and the zero coefficients are all in the lower left). When we use the substitution method on an upper triangular system, it is called *back substitution* (because we find the variables in backwards order).

This method generalizes much better when we start solving larger systems. For example, to solve the 3×3 system given at the beginning of this section, we want to eliminate (i.e., turn to zero) all the coefficients in the lower left. That is, we want to make a system of the form

$$\begin{cases} ax_1 + bx_2 + cx_3 &= d, \\ \mathbf{0x_1} + ex_2 + fx_3 &= g, \\ \mathbf{0x_1} + \mathbf{0x_2} + hx_3 &= i. \end{cases}$$

There are several approaches, but we will always follow a standard order. This is easier to see with an example.

First, we will eliminate the x_1 coefficient from the second equation. We can do this by multiplying the first equation by -2 and adding it to the second equation. This gives us the new system

$$\begin{cases} 2x_1 + x_2 + x_3 &= 1, \\ 0x_1 + x_2 + x_3 &= -1, \\ 8x_1 + 7x_2 + 9x_3 &= -1. \end{cases}$$

Next, we will eliminate the x_1 coefficient from the third equation. We can do this by multiplying the first equation by -4 and adding it to the third equation. We obtain

$$\begin{cases} 2x_1 + x_2 + x_3 &= 1, \\ 0x_1 + x_2 + x_3 &= -1, \\ 0x_1 + 3x_2 + 5x_3 &= -5. \end{cases}$$

Finally, we will eliminate the x_2 coefficient from the third equation. We can do this by multiplying the second equation by -3 and adding it to the third equation. (Note that it would not be a good idea to multiply the first equation by -3 and add it to the third equation, because that would undo the work that we did in the last step.) We get

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

Now our system is in upper triangular form, so we can use back substitution to solve it. The third equation is easy to solve and gives us $x_3 = -1$. Plugging this into the second equation, we find that $x_2 = 0$. Plugging each of these into the first equation, we find that $x_1 = 1$.

The order in which we eliminated the coefficients is somewhat arbitrary, but it is important to be consistent so that we can turn this into code. (In addition, this particular order will make our life easier in the next section.) When we use this order (i.e., eliminate all of the x_1 coefficients, then all of the x_2 coefficients, etc.) the method is called *Gaussian elimination*. Hopefully you can see how this generalizes to larger systems, and if you can't then I encourage you to try experimenting with a 4×4 system until it becomes clear. However, the amount of arithmetic and writing rapidly becomes cumbersome as the number of equations increases. Fortunately, lots of arithmetic in a prescribed order is exactly what computers are good at.

Matlab has many builtin methods to solve systems of equations, but we first need to input our equations. To do so, we will rewrite our system as a matrix equation. We do this by putting all of the coefficients from our system in a matrix, putting all of the variables into a vector and putting all of the right-hand side values into a vector. For the first system, we have

$$A = \begin{pmatrix} 2 & 1 \\ -6 & 1 \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} x \\ y \end{pmatrix} \quad \text{and} \quad \mathbf{b} = \begin{pmatrix} 5 \\ -3 \end{pmatrix}.$$

Now it will become apparent why we defined matrix multiplication the way we did. We have

$$A\mathbf{x} = \begin{pmatrix} 2 & 1 \\ -6 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 2x + y \\ -6x + y \end{pmatrix} = \mathbf{b}.$$

That is, we can rewrite our whole system as the single equation $A\mathbf{x} = \mathbf{b}$. We can now define our system in Matlab by defining the matrix A and the vector $\mathbf{b}$. (We don't know how to define $\mathbf{x}$ in Matlab, but we don't need to since that is what we are trying to solve for.)

Similarly, the 3×3 system from above is equivalent to the matrix equation $A\mathbf{x} = \mathbf{b}$, where

$$A = \begin{pmatrix} 2 & 1 & 1 \\ 4 & 3 & 3 \\ 8 & 7 & 9 \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \quad \text{and} \quad \mathbf{b} = \begin{pmatrix} 1 \\ 1 \\ -1 \end{pmatrix}.$$

Once we have defined the matrix A and the vector $\mathbf{b}$ in Matlab, it is very easy to solve the system: We just type `x = A\b`. (That is the backslash symbol, and you should read that code as "A backslash b.") For the moment, we will pretend that backslash uses Gaussian elimination followed by back substitution. (It is not far from the truth, and we will discuss the differences next class.)

4 Speed of Gaussian Elimination

It is generally not enough to be able to solve a problem numerically; we also usually need our solution to be efficient. To figure out how efficient our method is, we need an estimate of how much work we actually have to do to solve a system. The usual approach with numerical methods is to give a rough estimate of how the work changes with the size of our problem. For linear systems, we will use the number of variables (or the number of equations - they are the same in our case) as the size. We will call this number N. So, for instance, the first example we looked at has $N = 2$ and the second example has $N = 3$. It is very common to have to solve systems with $N = 100$ or $N = 1000$, and often systems can be much bigger than that.

Detailed estimates of speed are certainly important, but they quickly become very difficult. One of the reasons for this difficulty is that the fundamental operations that your computer executes when you type a line of code can vary widely from one version of Matlab to another and from one computer to another. Since we can't possibly go into all of these details, our estimates will be quite rough. This means that we are free to ignore what might seem like substantial details (for instance, we will throw away factors of two whenever convenient), since these details will be affected by many other issues that we can't measure. Fortunately, our rough estimates will still prove very useful, and most mathematicians, computer scientists and programmers never have to worry about better estimates than these.

Before we can start, we need to know what fundamental steps our computer can execute. As mentioned above, this varies from computer to computer, but we will assume that our processor can multiply two numbers together and add them to another number. That is, if x, y and z are numbers (not vectors), then the processor can compute $xy + z$ in a single instruction[1]. Such an operation is called a flop (floating point operation). We will try to count (roughly) how many flops it takes to perform Gaussian elimination. Once we have that estimate, we can multiply by how long the computer takes to execute one flop and therefore find the total time for our algorithm.

Let's start with the elimination portion of our method. We have to zero out several coefficients. For each coefficient we have to multiply a row by some number and then add it to another row. That sounds like a multiply/add operation, but instead of operating on single numbers we are operating on an entire row. This means that we have to use N flops to zero out a single coefficient. (Actually, we need $N + 1$ because we have to change the right hand side as well, but we'll ignore the extra 1.) How many coefficients do we have to zero out? We need to eliminate everything below the diagonal, which is a little less than half of the total number of coefficients in the system, so a little less than $N^2/2$. We will just approximate this by N^2. (I told you this was a rough estimate!) This means that we need to zero out approximately N^2 coefficients and each one takes N flops. That means that the elimination process takes roughly N^3 flops in total. We say that the elimination process is $\mathcal{O}(N^3)$, which you can read as either "order N cubed" or "big-oh of N cubed". The $\mathcal{O}$ lets you know that we ignored constant factors and smaller powers of N, but that the cubic power is correct.

What good is this estimate? If we knew how long it took to execute a flop, we could then estimate how long it would take to do Gaussian elimination. Let's say it takes t seconds to execute a flop. The total time for the whole algorithm is therefore approximately tN^3.

The key thing to notice here is that the t in our total time has a power of 1, while the N in our total time is raised to a power of 3. This means that if you get a computer that is ten times as fast (so t is ten times smaller), then your algorithm will take ten times less time to run. However, if you make your problem ten times as big (so N is ten times larger), then your algorithm will take $10^3 = 1000$ times longer. For instance, if $t = 0.000001$ seconds, then a 100×100 system would take $0.000001 \cdot 100^3 = 1$ second to run, while a 1000×1000 system would take $0.000001 \cdot 1000^3 = 1000$ seconds, or about 16 minutes. A $10,000 \times 10,000$ system would take almost twelve days. As you can see, the cubic power means that our run

[1]In some computers, $xy + z$ takes two steps - one to multiply x and y and another to add z. However, most modern processors can do this in one step.

time gets out of hand quite quickly.