# Lecture 8 Notes

## 1 Introduction

Last week, we discussed two related methods for solving systems of equations: Gaussian elimination (coupled with back substitution) and $LU$ decomposition (coupled with forward and back substitution). We found that forward and back substitution are fairly fast, but only work if we have a triangular system. $LU$ decomposition is also fairly fast if we want to solve similar systems (i.e., with the same left hand side) repeatedly, but both $LU$ decomposition and Gaussian elimination are both quite slow (and indeed are exactly the same) if we only want to solve a system once. This leaves us at a bit of an impasse if we want to solve a large system once. This week, we will discuss a different class of solution techniques called *matrix splitting methods*. In particular, these techniques will all be iterative methods. That is, instead of finding the exact answer (barring rounding error) in some fixed number of steps, they keep improving an answer but usually take an infinite number of steps to get an exact solution. This may sound like a problem, but in practice we generally only need a solution with a few (1-15) digits of accuracy, so we only need to take a finite number of steps. Our hope (and this will often be born out in practice) is that we will only need a few steps to find a good approximate answer, and so these methods will be substantially faster than Gaussian elimination.

## 2 Matrix Splitting Methods

Recall from the last lecture that $LU$ decomposition involved factoring a matrix into two simpler components. In particular, we found $L$ and $U$ such that $A = LU$. This was beneficial because $L$ and $U$ were triangular, so we could solve systems of the form $L\mathbf{x} = \mathbf{b}$ and $U\mathbf{x} = \mathbf{b}$ fairly quickly. In this section we will follow plan that is similar in spirit, but quite different in implementation. Instead of factoring $A$ as the *product* of two matrices, we will split $A$ into the sum of two matrices. In particular, we will write

$$A = P + T,$$

where $A$, $P$ and $T$ are all $n \times n$ matrices. In particular, we will insist that $P$ be in some "nice" form. That is, we should be able to solve equations like $P\mathbf{x} = \mathbf{b}$ in $\mathcal{O}(n^2)$ flops or less. (Also note that the name $P$ has nothing to do with the permutation matrices of the last section; it is just an arbitrary name.) Unlike with

$LU$ decomposition, it is very easy to find a matrix splitting like this. Once you decide what $P$ is (and that choice is entirely up to you) we know that $T = A - P$.

How can we use this splitting to solve a system of equations? Suppose we start with $A\mathbf{x} = \mathbf{b}$. We have

$$\begin{aligned} A\mathbf{x} &= \mathbf{b}, \\ (P+T)\mathbf{x} &= \mathbf{b}, \\ P\mathbf{x} + T\mathbf{x} &= \mathbf{b}, \\ P\mathbf{x} &= -T\mathbf{x} + \mathbf{b}. \end{aligned}$$

By design, it should be easy to solve this last equation (assuming we know the right hand side), so we can write this as

$$\mathbf{x} = P^{-1}\left(-T\mathbf{x} + \mathbf{b}\right) = -P^{-1}T\mathbf{x} + P^{-1}\mathbf{b}. \tag{1}$$

(Remember, $\mathbf{x} = P^{-1}\mathbf{c}$ is just another way of writing "the solution to $P\mathbf{x} = \mathbf{c}$". You should always mentally translate this to `x = P\c`.)

The problem with this method is that we haven't actually successfully solved for $\mathbf{x}$: There is still an $\mathbf{x}$ on the right hand side of our equation. If we did know the actual solution $\mathbf{x}$, then equation (1) would certainly be true, but it doesn't seem like it helps us solve anything. Fortunately, we can solve this by using an iterative approach.

Our reasoning is as follows: If we did know the real value of $\mathbf{x}$, then there would be no problem computing the right hand side of (1) (and indeed it would be exactly the same as the left hand side). We will therefore pretend that we know this value and substitute a guess for $\mathbf{x}$. We will call this guess $\mathbf{x}_0$. If we substitute this guess into the right hand side, we obtain a new value for $\mathbf{x}$. This new value is probably not the correct answer, but one might hope that it is closer than our original guess. We will call this new value $\mathbf{x}_1$, so we have

$$\mathbf{x}_1 = -P^{-1}T\mathbf{x}_0 + P^{-1}\mathbf{b}.$$

We then repeat this process to obtain a new guess:

$$\mathbf{x}_2 = -P^{-1}T\mathbf{x}_1 + P^{-1}\mathbf{b}.$$

We can continue this process as long as we want. In general, if we already have the $(k-1)$st guess $\mathbf{x}_{k-1}$, then the $k$th guess is given by

$$\mathbf{x}_k = -P^{-1}T\mathbf{x}_{k-1} + P^{-1}\mathbf{b}. \tag{2}$$

Two obvious questions arise: First, when do we stop? Obviously we can't repeat this process forever, so we need some sort of stopping criterion. Second, how do we know that this process works? It is not at all obvious (and often not at all true) that the guesses ever get close to the true solution. We need some sort of test to determine whether or not this method will converge.

## 2.1 Stopping Criterion

To determine when we will stop our procedure, notice the following fact: If our $(k-1)$st guess is exactly equal to the true solution (that is, $\mathbf{x}_{k-1} = \mathbf{x}$), then the $k$th guess will also be equal to the true solution. This follows immediately from equation (1). Therefore, if we ever find the correct answer then $\mathbf{x}_k = \mathbf{x}_{k-1}$, so $\mathbf{x}_k - \mathbf{x}_{k-1} = 0$. In reality, we will probably never find the actual solution, which means that $\mathbf{x}_k - \mathbf{x}_{k-1}$ will never be exactly zero. However, if our guess is close to the true solution, then $\mathbf{x}_k - \mathbf{x}_{k-1}$ will be close to zero.

There is one extra wrinkle here: $\mathbf{x}_k - \mathbf{x}_{k-1}$ is a vector, not just a number. We really want all of the entries to be small, but it isn't very wise to write something like `if (x2 - x1) < tolerance` in Matlab, because comparisons with vectors do not necessarily work the way you would expect. Instead, we will check something called the norm of $\mathbf{x}_k - \mathbf{x}_{k-1}$. When written by hand, we denote the norm by $\|\mathbf{x}_k - \mathbf{x}_{k-1}\|$. In Matlab, it is `norm(x2 - x1)`. If you are familiar with polar or spherical coordinates, the norm of a vector is the same thing as the radius/magnitude. We will also occasionally use the command `norm(x2 - x1, Inf)` instead, which is denoted by $\|\mathbf{x}_k - \mathbf{x}_{k-1}\|_\infty$. This just uses a slightly different formula, but still gives a notion of "how big" the vector is.

We will therefore stop iterating once $\|\mathbf{x}_k - \mathbf{x}_{k-1}\|$ is less than some small predefined tolerance.

## 2.2 Convergence Criterion

The more difficult question is: How do we know if our guesses will approach the correct answer? It seems entirely possible that the guesses $\mathbf{x}_k$ will all be bad guesses and we will never find the true solution. To answer this problem, we will define the error of our guess as $\mathbf{e}_k = \mathbf{x}_k - \mathbf{x}$. If $\mathbf{e}_k$ is all zeros (remember, it is a vector) then our $k$th guess will be exactly the same as the true solution, which would mean that we found the correct answer. Similarly, if all of the entries of $\mathbf{e}_k$ are very close to zero then our $k$th guess will be very close to the true solution. Before we can use these matrix splitting methods, we want some sort of guarantee that the errors $\mathbf{e}_k$ will go

to zero as we make more guesses.

We already know that

$$\mathbf{x}_k = -P^{-1}T\mathbf{x}_{k-1} + P^{-1}\mathbf{b} \text{ and}$$
$$\mathbf{x} = -P^{-1}T\mathbf{x} + P^{-1}\mathbf{b}.$$

If we subtract these two, we obtain the equation

$$\mathbf{x}_k - \mathbf{x} = -P^{-1}T\left(\mathbf{x}_{k-1} - \mathbf{x}_k\right).$$

Rewriting this in terms of the errors $\mathbf{e}_k$ and defining $M = -P^{-1}T$ for convenience, we find that

$$\mathbf{e}_k = M\mathbf{e}_{k-1}.$$

If we plug in $k = 1$ we find that

$$\mathbf{e}_1 = M\mathbf{e}_0.$$

Similarly, if we plug in $k = 2$ we find that

$$\mathbf{e}_2 = M\mathbf{e}_1 = MM\mathbf{e}_0 = M^2\mathbf{e}_0.$$

Likewise,

$$\mathbf{e}_3 = M\mathbf{e}_2 = MM^2\mathbf{e}_0 = M^3\mathbf{e}_0.$$

The pattern should become obvious after a few steps:

$$\mathbf{e}_k = M^k\mathbf{e}_0.$$

This is a very useful formula. In a real problem, we will not know what $\mathbf{e}_0$ is (because it depends on the true solution $\mathbf{x}$), but we will know $M$. There is hope that we can show that $\mathbf{e}_k$ goes to zero without ever having to find $\mathbf{e}_0$. To see why this is promising, we will first ignore the fact that $M$ is a matrix and the $\mathbf{e}$'s are vectors. If we pretend that all three are simply numbers, then our analysis becomes very easy. Notice that if $M$ is larger than 1 (either positive or negative) then $M^k$ gets larger and larger as $k$ increases. Likewise, if $M$ is smaller than 1 (either positive or negative) then $M^k$ gets smaller and smaller as $k$ increases. This immediately gives us the rule we were looking for (again assuming that $\mathbf{e}_k$, $\mathbf{e}_0$ and $M$ are numbers):

- If $|M| > 1$, then $\mathbf{e}_k$ goes to infinity as $k$ goes to infinity.
- If $|M| < 1$, then $\mathbf{e}_k$ goes to zero as $k$ goes to infinity.

This means that our iterative method will find the right solution if $|M| < 1$ and will fail completely if $|M| > 1$. (We will not worry about the case where $M = 1$. It is easy in the case where $M$ is a single number, but it does not generalize nicely when $M$ is a matrix.) We can find $M$ easily before starting our algorithm, so it is easy to check if the method will converge.

Our next goal is to generalize this idea to the full problem where $M$ is a matrix and the $\mathbf{e}$'s are vectors. It will turn out that we get almost the same rule, but we will need some more mathematical machinery before we can see why.