Week 3 Part 3: Computational Complexity

This won't be a large focus of the course, but it is worth mentioning.

Matrix Multiplication

Consider the multiplication of $A_{l\times m}B_{m\times n}$. To multiply we take the dot product of the rows of A with the columns of B,

$$\begin{bmatrix} --- & a_1 & --- \\ --- & a_2 & --- \\ & \vdots & \\ --- & a_l & --- \end{bmatrix} \begin{bmatrix} | & | & & | \\ b_1 & b_2 & \cdots & b_n \\ | & | & & | \end{bmatrix} = \begin{bmatrix} a_1\cdot b_1 & a_1\cdot b_2 & \cdots & a_1\cdot b_n \\ a_2\cdot b_1 & a_2\cdot b_2 & \cdots & a_2\cdot b_n \\ \vdots & \vdots & \ddots & \vdots \\ a_l\cdot b_1 & a_l\cdot b_2 & \cdots & a_l\cdot b_n \end{bmatrix}$$

When we analyze the number of steps required, we observe that each entry requires m multiplications and $m-1$ additions to carry out the dot product. This gives us $2m-1$ operations. However, there are $l\times n$ entries, so it takes (2m-1)ln operations. Without loss of generality, suppose $n > l, m$, then $(2m-1)ln \leq 2n^3 \sim O(n^3)$. This is called computational complexity. Notice that we don't care about constants.

For Gaussian elimination the computational complexity is $O(n^3)$ as well. It also makes no difference if you augment or use $Lc = b$.

These computational complexities are for dense matrices, however sparse matrices have faster algorithms associated with them. If a matrix has a lot of zeros, then the operations become a lot easier. A special type of matrix that appears in applications is called a tridiagonal matrix.

Lets look at a slightly different problem for computational complexity. Lets think about how we sort numbers from smallest to largest. There are many different types of sorting algorithms, but perhaps the most natural is the Selection sort, where we simply look for the smallest value and move it to the front of the sequence. This will have a computational complexity of $O(n^2)$. If we try a more sophisticated sort (say Quick sort), then we look at two adjacent numbers and ask which one is smaller. If the number to the left is smaller, we don't do anything. If the number to the right is smaller, we switch them. This decreases the amount of operations we have to do since we are only comparing two adjacent numbers and not a single number to the whole list. This algorithm has a computational complexity of $O(n\log n)$.