Welcome to Numerical Analysis, the study of computation in a finite-precision environment. Numerical Analysis is that branch of mathematics that tells us how fast numerical algorithms are (and, sometimes, with fast algorithms solving a particular problem could be), and it also tells us how stable algorithms are, meaning, whether or not they will, in certain situations, return “garbage” outputs due to accumulation of numerical (roundoff) error. It will give you the tools and algorithms to perform good and reliable computation, in addition to knowing which of them work best under what circumstances.

The first quarter of the sequence is all about numerical linear algebra. We will learn that there are, roughly, two main problems in classical Linear Algebra: finding ways to solve $Ax = b$ well (and that could include least squares problems, where, rather than downright solving, we will try to find the $x$ such that $Ax$ “best approximates” $b$), and finding ways to reliably calculate $(x, \lambda)$ for which $Ax = \lambda x$. In the above, $A$ is a matrix, $x$ is a vector, $\lambda$ is a constant.

On our journey, we will touch on topics that you have seen before, in Math 18 or equivalent prerequisites, like Gaussian elimination and the Gram-Schmidt process, but also notions that you will likely see here for the first time, like iterative solving methods, condition numbers, and the singular value decomposition. We will think about algorithms, but also about what makes them good algorithms. Word of caution: although we are likely to mostly deal with notions that are familiar in the first week or two, do not make the mistake of thinking that the course will be easy. If you do, Midterm 1 will hit you like a ton of bricks.

There is much that we will not talk about, although I will from time to time (maybe often) suggest to you reading and extra (ungraded) problems that will go a little deeper into the heart of the subject.

Math 170A is a challenging course, as it involves both theoretical and computational aspects of linear algebra; I hope it will give you an idea of the challenges of working in a constrained, finite-precision environment. I welcome you to this course, as I believe that, ideally, any student interested in applied math, statistics, computer science, engineering, and basically any kind of science should be taking an introductory numerical analysis course before they graduate, so that they learn neither to write code blindly, as if their computational environment had infinite precision, nor to blindly use black-box code. To know an algorithm is to know its limitations in a given computational environment.

We will now discuss Administrivia; please refer to the copy of the syllabus I have handed out. This is a short version of a more detailed pile of information that can be found on the course website,

www.math.ucsd.edu/~dumitriu/math170a.html

A couple of words about your instructor: I am new at UCSD, although not new to teaching (I spend more than a decade as a professor at University of Washington, where I moved here from last summer). I have taught linear algebra and numerical linear algebra before, although this class size is a bit larger than what I am used to. I don’t know everything about “how things work at UCSD” just yet, so I will be doing some learning alongside you.

Discussion of syllabus.
And now, back to numerical linear algebra. Today’s agenda will be to talk a little about simple computation, and its complexity.

As we go through the course, certain things will be taken for granted because they are contained in the prerequisites; for example, today I will talk about vectors and matrices, vector-vector, matrix-vector, and matrix-matrix multiplication, and assume you know what I am talking about.

As I mentioned, we will be using MATLAB a lot, and one of the things you need to know is how to introduce and initialize a column vector of length $n$ in MATLAB (call this $v$):

\[
\text{>> } v = \text{zeros}(n,1);
\]

Suppose we want to do a very simple computation in MATLAB, like for example multiplying two vectors of length $n$. We might want to write a function which takes as input a number $n$ and two column vectors, each with $n$ components, and returns the (inner) product of these vectors.

Here is the simplest way to do this in MATLAB, with the help of a for loop.

```matlab
function product = vecvec(n, u, v);
    if length(u) ~= n,
        error('u has the wrong format');
    end;
    if length(v) ~= n,
        error('v has the wrong format');
    end;
    product = 0; % initializes the product; this is how we comment out things
    for i = 1:n
        product = product + u(i) * v(i);
    end
end
```

How fast is this code? How much longer will it take to run if we let $n = 100$? $n = 200$? $n = 800$?

One way to predict this is to count how many operations (+−, ∗/) the algorithm does. (In reality, many systems spend a lot more on multiplication/division than one addition/subtraction, so sometimes only the former are counted, but here we will count everything.)

Each execution of the for loop has 2 operations, and there are $n$ executions, so the answer is $2n$. Therefore the cost is linear.

It will take roughly twice as long to perform the same calculation if $n = 200$, 8 times as long if $n = 800$, and 200 times as long if $n = 20,000$. (If we increase $n$ too much, other issues will arise, like the cost of transporting data from the slow, storage memory to the fast, computing memory, but this is the subject of a different kind of course.)

For HW 0, you will have to do the same exercise for matrix-vector, respectively, matrix-matrix multiplication, and count the number of operations performed.