Math 4610 Fundamentals of Computational Mathematics - Single and Double Precision Numbers.

As a way to make sure that finite precision numbers and the arithmetic operations used to work with these numbers produce correct results we need a consistent definition for these numbers. It is typical to define numbers in terms of a fixed format that can be manipulated in predictable ways. A standard way to classify the storage of numbers is in terms of whether the number representation is to be single or double precision. Single precision is a number format that occupies 32 bits and double precision is a number format that occupies 64 bits. Note that the use of these formats also dictates memory in terms of the total amount of memory and disk space available on a computer.

Note that for integers the story is a little different. Integers can be represented exactly with a given number of bits. For example, if we work with a signed 32 bit integer, the largest integer that can be represented in memory is $2^{31} - 1 = 2,147,483,647$. For real numbers, we need more information to be able to represent a number. For single precision numbers the IEEE 754 representation is defined by 32 bits organized into the following

```
- bits 0 through 22 (23 bits) define the mantissa or fraction
- bits 23 through 30 (8 bits) define the integer exponent
- bit 31 (1 bit) for the sign of the number
```

So, a number can be written in the form

$$(-1)^{b_{31}} \times 2^{b_{30}b_{29}\cdots b_{32}-127} \times (1.b_{22}b_{21}\ldots b_0)_2$$

Note that all the digits, b_i, $i = 0, 1, \ldots, 31$ are binary digits. These digits are either 0 or 1. Also, the exponent is shifted to be symmetric about the origin. This is used to resolve very small numbers (e.g, 2^{-127}) and very large numbers (e.g, 2^{128}).

In the last part of this lecture, an algorithm was coded that returns the number of digits expects to correct for a machine/computer number. We can actually verify that this makes sense as follows.

- $1 + 2^{-23} \approx 1.000000119$
- $2 - 2^{-23} \approx 1.999999881$
- $2^{-126} \approx 1.175549435 \times 10^{-38}$
- $2^{127} \approx 1.70141183 \times 10^{38}$

The importance of these numbers is the following. The first number is exactly the value we should see from the single precision machine epsilon code.

With this finite discrete representation there is a limit as to the set of numbers that can be represented.

Any work that is done on a computer boils down to manipulating numbers. A problem with this is that computers have finite resources and the representation of many numbers requires the use of an infinite number of decimal digits. For example, given a circle, the formula for the circumference is

$$C = 2 \times \pi \times r = \pi \times d$$

where r is the radius of the circle and d is the diameter of the circle. The number π is not a rational number. That is, the decimal expansion of this value has an infinite fractional part. The value can be represented as follows:

$$\pi \approx 3.141592653589793...$$

where the ellipsis notation, ..., means the digits never repeat. So, to get an exact representation of π it is necessary to have an infinite number of digits available. Since computer resources are finite, we must settle for an approximation.

In this part of the lecture, we will use a few examples that should motivate us to spend some time on this issue and more fully understand the implications of finite precision of number representation.

For the first example, we could use the approximation

$$\pi \approx 3.141592653589793$$

without including an infinite number of digits. One question that should arise is how many digits will provide us with an accurate enough approximation. One of the programs used over the past few decades to " burn in machines was an algorithm to compute more and more digits of π. This means that is possible to determine π to any degree of accuracy that we want. However, it is not practical for real problems.
In some cases, a very crude approximation is enough. In some of our United States, laws have been passed to legally approximate π using a rational number. For example,

$$\pi \approx \frac{22}{7}$$

provides an approximation that will hold up in a courts of law. If you are pouring a circular concrete slab for a water tank it is a good idea to have an estimate of the amount of concrete based on an accepted value for the number π.
Basically, numbers are best represented on a computer using zeros and ones - or in a binary number system. Other common number systems used in computer architecture/hardware are in octal (or base 8) and hexadecimal (or base 16). Another issue that arises in the representation of numbers is numbers that are relatively prime to base 2. As a simple example, consider the representation of the number 1/3 in base 2. The value is

$$\frac{1}{3} = 0.01010101.....$$

where the last pair of digits repeats forever. If a finite number of binary digits are used to represent 1/3, the result is an approximation of the exact value. Note that a base 10 representation of 1/3 is given by the decimal representation

$$\frac{1}{3} = 0.3333333333333.....$$

Even if computers worked in a base 10 system, we would necessarily have to settle for approximate number representation.
Since there are an uncountable number of irrational numbers, it is impossible to imagine a computer that would not suffer the same issue. So, the best we can hope for is that there is an accepted number representation that will work on all computers. There is a standard (IEEE standard reference here) for number representation that we will look into later in the course. For now, we will assume that all of the computers we will use will behave the same way. From a practical point of view, it would be nice to be able to compute the limits of the accuracy of machine numbers. Fortunately, we can write a little program that will do the trick for us.