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/accepted format for numbers used on any/all computers. It is typical to define numbers in 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 of single or double precision. Single precision is a number format that occupies 32 bits and double precision is a number format that typically uses 64 bits. Note that the use of these formats also dictates memory limitations in terms of the total amount of memory and disk space available to actually do work.

Note that for integers the story is a little different. Integers can be represented exactly given enough 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. Real numbers are typically described by (1) the mantissa or fractional part of a number, (2) an (integer) exponent for the base being used, and (3) a plus or minus sign represented by a zero for a positive number or a one for negative numbers.

Given these considerations for the format of computer numbers and a specified number of bits to make available for numbers, the only thing left is to determine how to divide up the bits. It makes sense to use the same number of bits for each number used on a computer. For single precision numbers the IEEE 754 standard for this representation is defined by 32 bits organized as follows.

- 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 standard single precision number can be written in the form

$$x_{base2} = (-1)^{b_{31}} \times 2^{b_{30}b_{29}\cdots b_{32}-127} \times (1.b_{22}b_{21}\ldots b_0)_2$$

Note that the terms, 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. That way the exponent can be used to resolve very small numbers (e.g, 2^{-127}) and very large numbers (e.g, 2^{128}). This gives a complete description of the 32-bit format. Note that the bits are indexed from zero and not one.

In a previous part of this lecture, an algorithm was coded (maceps) that returns the number of digits expected to be correct for a machine number. We can actually verify that this makes sense using the following list of values in 32-bit numbers.

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

The importance of these specific numbers is the following. The first number is exactly the value we should see from the single precision machine epsilon code before the number 1 is subtracted. The number of digits of accuracy is then available using the fractional piece of the approximation.

The term double precision refers to a 64 bit format with the bits distributed as described below.

- bits 0 through 51 (52 bits) define the mantissa or fraction
- bits 52 through 62 (11 bits) define the integer exponent
- bit 63 (1 bit) for the sign of the number

More bits in the mantissa and exponent means exact represntation of more real numbers. Note tha there will still be lots of holes in the numbers represented by a 64-bit machine number. The machine epsilon code should indicate the accuracy of the approximation just as in the 32-bit case.

Computers would be incredibly useless if all a computer could do is store approximations of numbers. What makes a computer extremely useful is the ability to combine numbers through arithmetic operations and do the work accurately (no human error) and efficiently. In this part of the course we will use the notation

$$fl(x) \approx x$$

where fl is associated with the term floating point. Most operations done by computers are the four standard binary operations of addition, subtraction, division, and multiplication. The term floating point comes from process of lining up the decimal points before performing the operation. This is done with a shify in the exponent of the numbers.
For the moment, suppose that we have two numbers and want to look at the output of a single binary operation. Without loss of generality, we can assume the numbers are both positive. If the numbers are x and y, then we can write

$$x = fl(x) + \epsilon_x$$

and

$$y = fl(y) + \epsilon_y$$

Adding the two results gives

$$x + y = (fl(x) + \epsilon_x) + (fl(y) + \epsilon_y) = (fl(x) + fl(y)) + (\epsilon_x + \epsilon_y)$$

Since the values are positive, one would expect that the error in this case will be small.
The binary of operation of subtracting two numbers is not as stable. For

$$x - y = (fl(x) - \epsilon_x) + (fl(y) - \epsilon_y) = (fl(x) - fl(y)) + (\epsilon_x - \epsilon_y)$$

The problem is that if the two numbers are within machine precision of each other, then the difference will produce a value that is of the same order of magnitude as the difference

$$\text{error } = \epsilon_x - \epsilon_y$$

In floating point arithmetic, the two numbers are aligned at the decimal point and when the difference is computed all significant digits can canel out. Since there are no extra digits to use in these representations, it is usual that the output from this operation is garbage. This is called catastrophic cancellation. The real issue is if a result like this is used in a later computation. This means the output cannot be trusted.
Multiplication and division behave similarly to addition and subtraction, respectively. That is. the multiplication of two numbers is a stable operation while the ratio of two numbers is not stable for all computations. We can write the product of two numbers as

$$x * y = (fl(x) + \epsilon_x) * (fl(y) + \epsilon_y) = fl(x) * fl(y) + \epsilon_x fl(y) + \epsilon_y * fl(x) + \epsilon_x * \epsilon_y$$

and for the ratio of numbers

$$\frac{x}{y} = \frac{fl(x) - \epsilon_x}{fl(y) - \epsilon_y}$$

The expression for the product is not too difficult to analyze. The last three terms in the expansion of the product are, in general, small relative to the first term. So, the product is relatively stable. As for the ratio of numbers, the analysis is a bit more complicated. A typical way to handle these types of errors is to use interval analysis. We will take up an introduction to interval analysis in the near future. For now, we will move on to the types of error one encounters in using computers to solve problems.
From the start there have been cases where machine precision errors have caused disasters. There are a couple of web sites that document these types of failures The sites can be found at:

`http://www.ima.umn.edu/~arnold/disasters`

and

`http://wwwzenger.informatik.tu-muenchen.de/persons/huckle/bugse.html`

It is informative to check these types of cautionary tales so that we do not reproduce the same types of errors. There is also a book one can read entitled " Set Phasers on Stun" with similar types of real world problems that were caused by not paying attention to the details.
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.