This is a working set of notes for in-class discussion (APPM 4600) on an introduction to what numerical methods and numerical analysis are (the subject matter of our course) and then mainly on introducing the concepts needed to do arithmetic with real numbers in a computer, known as *floating point arithmetic*.

1 What are numerical methods?

There are basically three ways a scientific question or problem can be studied:

1. **Experimentally:** Design and perform an experiment (repeatedly and independently to try to replicate it).
2. **Analytically:** We make a mathematical-logical model, and then solve this model by "pen and paper".
3. **Numerically:** We make a mathematical-logical model, and then simulate / approximate the answer with a computer.

Experiments can be impossible to carry out or to replicate, and even if possible, they can be expensive and risky in many ways. To make matters worse, many mathematical problems of interest just cannot be solved in closed form via pen-and-paper (e.g. N-body problem).

As our computational power has increased (especially after the advent of computers), the third option on this list (that is, numerical methods and scientific computing) has emerged as a tremendously useful third pillar for scientific discovery and technological advancement. Numerical methods allow us to answer questions and perform simulations given fairly complex mathematical models, which effectively allows us to make "simulated scientific experiments". They have increasingly also played a role in data generation and training for ML algorithms, tools for experimental design and for smart, additive industrial manufacturing.

There are innumerable areas of application for numerical methods. Some notable ones include:

1. N-body problems (astrophysics, electrostatics, hydrodynamics, materials science)
2. Control theory (robotics, power plant, power grid simulation and stability)
3. Industrial design (e.g. smart materials, stealth planes, scattering, medical imaging)
4. 3D printing (mechanics, shape optimization)
5. Biophysical and biomedical applications
6. ...

In this course, we will be covering some of the main building blocks that undergird state-of-the-art numerical methods for scientific computing and simulation. We hope that this will serve as a solid basis for you to become a user and even a developer of better and faster ways to answer scientific questions.

2 Floating Point Arithmetic

2.1 How do we do math in a computer? How do we represent and work with real numbers?

For most of us as students, our exposure to solving mathematical problems and working with numbers happens in the realm of "mathematics with pen-and-paper": we are taught to massage mathematical expressions until the answer or answers come out. This can be assisted by computational tools, from calculators to wolfram alpha, but we tend to treat them as black-box assistants in figuring out what the answer and logical steps to it have to be.

More specifically, when we work with real numbers, we are taught that every real number is determined by its expansion on a given basis (we tend to use decimal), and that the real numbers as a set (or as part of the real number line) have the following interesting properties:

1. **Infinite (uncountable)** - there are an uncountable infinity of real numbers, and in fact, an uncountable infinity of them in any closed interval like $[0, 1]$.
2. **Dense** - between any two distinct real numbers, there is another real number.
3. **Unbounded** - there is no biggest (in absolute value) or smallest non-zero (in absolute value) real number.
4. **Uniform** - If you generate a number at random in $[0, 1]$, the likelihood that it is less than $1/2$ is 50% (0.5).

We are also taught at least two notations for them: the usual one writing whatever decimal digits we know (all non-zero to the left and as many to the right as we can muster) and scientific notation (e.g. 6.022×10^23).

2.1.1 Building a finite precision arithmetic system

Now, imagine we go back to the dawn of computing, and we are tasked with implementing real numbers and basic arithmetic operations with them. How would we do this? The first important thing to note is: *we do not have infinite memory.* And so, we must settle on a *memory budget* for each real number. How many bits can we spare per real number?

Let's say we decide each real number gets 64 bits. That is, we have to use 64 binary digits (0's and 1's) to represent our real number. This means that, inevitably, the numbers in our system will have **finite precision**: the computer will only be able to know *some limited amount* of digits for them.

In class, together we discussed how to best allocate these 64 bits. We opted to use binary representation, and to write numbers as

$$x = \pm(d_1 d_2 \cdots d_{32}.d_{33} \cdots d_{64})_2$$

where the point between the 32 and 33 digits marks where positive powers of 2 stop and negative powers start (much as it does for our decimal system). We then talked about what strengths and weaknesses this system could have, mainly in terms of the potential *absolute and relative* rounding errors we would make by simply writing down a number down in the computer. This system would favor having a uniform bound on **absolute rounding error**, which we agreed is not very good for practical use.

We then discussed a system allocating these bits to favor **relative rounding error**. Given a base β (binary, or $\beta = 2$), we allocate one bit for the sign, k significant digits (for the so-called mantissa) and $64 - k - 1$ digits for the exponent. That is,

$$x = \pm(0.d_1 d_2 \cdots d_k)_2 \times 2^e$$

where

1. $d_1, d_2, \cdots, d_k$ are known as the significant digits, with $d_j \in 0, 1$ and $d_1 = 1$ (not allowed to be equal to 0).
2. $(0.d_1 d_2 \cdots d_k)_2$ is known as the mantissa.
3. $e \in [m, M]$, where usually if we have p digits for the exponent, e is an unsigned integer. This means it would go from 0 to $2^p - 1$. However, this number is understood in a *biased form* by subtracting $2^{p-1} - 1$, and the exponents of all ones or all zeros are reserved for special numbers.
4. **All zeroes** denote 0 (if the mantissa is zero) and subnormal numbers (if the mantissa is not zero)
5. **All ones** denote $\pm\infty$ (if the mantissa is zero) and NaN (if the mantissa is not zero).

You could denote this floating point number system as $\mathcal{F}(\beta, k, p)$.

We then used a toy system with $\beta = 2, k = 4, p = 3$ (8 bits total) to illustrate how this works. We concluded that

1. There is a maximum number representable to full precision.
2. There is a minimum non-zero number representable to full precision.
3. There is a finite, constant number of floating point numbers between powers of 2 ($2^{k-1} = 8$)
4. There is a finite number of floating point numbers in total. ($6 \times 8 = 48$ numbers plus zero and special numbers like $\pm\infty$.)

Finally, we discussed the two main floating point number systems used in scientific computing: IEEE 754 Double precision numbers (Float 64 bit) and Single precision numbers (Float 32 bit). We centered our discussion on the double-precision format, as it is the one that is most widely used and that we will be employing for the rest of our work in this class.

The Double precision format uses 64 bits, out of which 1 is for the sign, 52 are assigned to the mantissa, and 11 are assigned to the exponent. That means:

1. There is a maximum number double precision number, around 10^{308}.
2. There is a minimum non-zero number representable to full precision, around 10^{-308}
3. There is a finite, constant number of floating point numbers between powers of 2 ($2^{k-1} = 2^{51}$)
4. There is a finite number of floating point numbers in total. (2046×2^{51} numbers plus zero and special numbers like $\pm\infty$.)

Most importantly, we talked about what the absolute and relative errors are like for representing numbers in this system. That is

Definition 1 *Given a floating point system (e.g. double) and $x \in \mathbb{R}$, we denote its representation as $\text{fl}(x)$. The absolute and relative rounding errors resulting from this representation are given by:*

$$\text{err}_{abs}(x) = |x - \text{fl}(x)|$$
$$\text{err}_{rel}(x) = \frac{|x - \text{fl}(x)|}{|x|}$$

We first looked at the "distance" between two consecutive numbers for $e = 0$ and concluded that it is $2^{-52} \simeq 2.2 \times 10^{-16}$, and what is then the smallest possible absolute and relative (they are about the same here) errors due to rounding.We then took this to an example where $e = 10$. Here, clearly, the distance between real numbers is much higher (and so, the worst absolute error we can make is higher as well). However, the bound on the relative error is uniform!

We define the concept of the **Machine epsilon**: it is the distance between consecutive numbers in our system for $e = 0$, and a uniform relative rounding error bound:

Definition 2 (Machine epsilon) *Given a floating point system, we define the machine epsilon ε_{mach} as the distance between consecutive floating point numbers in this system for $e = 0$. For this system, we have the uniform relative error bound*

$$\text{err}_{rel} = \frac{|x - \text{fl}(x)|}{|x|} \leq \varepsilon_{mach}$$

Depending on how we round, this bound can be multiplied by a factor of 2.

When we translate it to decimal, it tells us that *for double precision numbers, we can trust up to 16 decimal (or 52 binary) digits of relative accuracy.* For this reason, double precision arithmetic is strongly associated with 16 digits. Similarly, the machine epsilon for single precision floats is around 10^{-8}, and so, we can trust up to 8 decimal digits of relative accuracy if we are using single.

2.2 Loss of precision

So far, we have only talked about rounding errors resulting purely from representing a number in floating point. However, this is only the beginning. Once we have numbers in our system, we wish to operate with them, to perform arithmetic such as sums, subtractions, multiplications, divisions, and other higher order operations (which in a sense boil down to a lot of the basic arithmetic operations).

An important question might then be: if we know numbers x, y to full precision in the double precision system (so, to 16 decimal digits of relative precision), is that also true of $x+y, x-y, xy, x/y$? Once we get a handle of that, we can then talk about the error for other more complicated function evaluations, like $sin(x-y)$ or $exp(xy)$, and in general, for any algorithm we may propose computing with real numbers.

Out of the four "basic" arithmetic operations, we find that addition of two numbers (of the same sign), multiplication and division (unless the numbers involved get very large or very small, to the point that they go beyond the smallest or largest float) preserve the digits of relative accuracy we could trust from their inputs. However, **subtracting two numbers of similar sign and order of magnitude** can cause significant loss of precision, and should be avoided if possible.

Examples

To see how this works, we can build some intuition by computing a few examples by hand using 5 decimal significant digit arithmetic. In every one of these, we must be careful to round to 5 significant digits *every time we perform any computation*.

- Let $\text{fl}(x) = 0.52348$ and $\text{fl}(y) = 0.52343$. In other words, we know the relative errors for each are bounded by about $0.00001 = 10^{-5}$. Add these numbers and determine how many digits you can trust and what the relative error is in this case.

  Answer: The answer computed to 5 digits is 1.0469. The correct answer is 1.04691, so the absolute error is 10^{-5}, and the relative error is 9.5×10^{-6}. We have lost no precision.

- Let $\text{fl}(x) = 0.52345$ and $\text{fl}(y) = 0.52343$. In other words, we know the relative errors for each are bounded by about $0.00001 = 10^{-5}$. Subtract these numbers and determine how many digits you can trust and what the relative error is in this case.

  Answer: The answer computed to 5 digits is 0.0002, or 0.2×10^{-3}. The absolute error is again about 10^{-5}, and so the relative error is 0.05. We have effectively lost 4 digits of accuracy (we only know one digit of our solution!).

- Do the same as above, but for multiplication and division.

  Answer: Left to the reader.

Example: solving a quadratic equation. Now, say we want to find the two roots for the quadratic equation

$$0.2x^2 - 47.91x + 6 = 0$$

The algorithm / recipe we probably all know to solve this is the infamous quadratic formula. The exact roots are then given by:

$$r = \frac{47.91 \pm \sqrt{47.91^2 - 4(0.2)6}}{2(0.2)} = \frac{47.91 \pm \sqrt{2295.3681 - 4.8}}{0.4}$$

To better see the loss of precision, we limit ourselves once again to doing the arithmetic with 4 significant decimal digits. This gives us:

$$r \simeq \frac{47.91 \pm \sqrt{2290}}{0.4} \simeq \frac{47.91 \pm 47.85}{0.4}$$
$$r_1 \simeq 239.4 \quad r_2 \simeq 0.15$$

Using double precision (or computing by hand), we would find the "true" solution to 4 digits is $r_1 = 239.4$ and $r_2 = 0.1253$.

What happened to the second root, and how can we fix it? Clearly, for r_2, we are subtracting two numbers that are close to each other, and for that reason, we lose $2 - 3$ digits. If we compute the relative error, we get an error of 0.1971.

When there is loss of precision, there are generally two ways to fix it.

- Brute force: we can *increase precision*, so that the loss of precision is not as big of a deal (e.g. if we start with 16 digits, losing 2 or 3 may not be that bad). This sort of patch is, however, not a good idea in the long run.
- Actual fix: change the algorithm so that you do not lose (as much) precision. In our case, since the subtraction is the problem, we use a trick from our calculus 1 class and multiply by the reciprocal:

$$\begin{aligned} r_2 &= \frac{-b-\sqrt{b^2-4ac}}{2a}\frac{-b+\sqrt{b^2-4ac}}{-b+\sqrt{b^2-4ac}} \\ &= \frac{b^2-(b^2-4ac)}{2a(-b+\sqrt{b^2-4ac})} = \frac{4ac}{2a(-b+\sqrt{b^2-4ac})} \\ &= \frac{2c}{(-b+\sqrt{b^2-4ac})} \end{aligned}$$

Applying this new algorithm to our example (remembering to use 4 digit arithmetic), we now find it gives us:

$$r_2 = \frac{12}{47.91+47.85} = \frac{12}{95.76} = 0.1253$$

We now lose no precision!

More examples In your first homework assignment, you are asked to use algebra and other identities (e.g. trigonometric identities) to change how you evaluate a function near a value of x where loss of precision is happening. That is:

1. $\sqrt{x+1}-1$ for $x \simeq 0$
2. $\sin x - \sin y$ for $x \simeq y$
3. $\frac{1-\cos y}{\sin x}$ for $x \simeq 0$

It is left to the reader to add the solutions to these exercises to their notes. We note this is far from the only way to evaluate these expressions to a desired target accuracy. After you solve these exercises using algebra or identities, try coming up with a different algorithm and compare the results! For example, for problems 1 and 3 above, you could try to use the Maclaurin expansions (Taylor expansions around $x = 0$) for the functions involved to come up with an approximation.