Numerical Mathematics 1: Numerical Analysis

Dr. Zahra Lakdawala[1]

[1] ***Associate Professor of Mathematics, LUMS University,***
zahra.lakdawala@lums.edu.pk

These notes have been compiled from various sources.

Abstract

1 Floating Point Arithmetic

1.1 First principle of floating point arithmetic

We usually perceive and understand the world in a discrete way. We define a quantity such as atoms and molecules as the smallest unit of ordinary matter in the world. In a similar way, when we deal with discrete notion of the continuous world in the mathematical world, we need to define the smallest unit. It is a well-known fact that there are infinitely many real numbers in the interval $[1, 2]$. There is no hope to store infinitely many points on a computer. We rely on the IEEE-754 double precision floating point standard that approximates the interval $[1, 2]$ by a discrete set of exactly $2^{52} \approx 4.5.10^{15}$ evenly spaced points/numbers, namely

$$1, 1 + 2^{-52}, 1 + 2.2^{-52}, 1 + 3.2^{-52}, \ldots, 2. \tag{1}$$

The discrete set of numbers that computers use to approximate $[1, 2]$ is unbelievably fine.

There are about $2.5.10^{25}$ molecules (at sea level) in a cubic meter of air. This means there are about 10^9 molecules per metre. However in a unit interval $[1, 2]$, there are $4.5.10^{25}$. This means that computers work on scale a million times finer than the physical world.

Similarly, the interval $[2, 4]$ is approximated again by 252 points:

$$2, 2 + 2^{-51}, 2 + 2.2^{-51}, 2 + 3.2^{-51}, \ldots, 4.$$

Thus since the interval length has increased, also gaps have grown. In general, the interval $[2^j, 2^{j+1}]$ is represented by 1 multiplied by 2^j .

The advantage of increasing the gaps for larger intervals is that these gaps are in a relative sense never larger than the so-called machine precision

$$\epsilon_{machine} = 2^{-52} \approx 2.2204 \cdot 10^{-16}$$

This leads us directly to the first principle of floating point arithmetic. Suppose we have some real number $x \in \mathbb{R}$ which is bounded away from $\pm\infty$ in such a way that we can approximate it with a finite double precision floating point number. We obtain this nearest neighbor floating point approximation denoted with $fl(x)$ via rounding. There are different rounding modes (round to nearest, up, down, toward zero). The most common method is round to nearest, which would round 3.5 as well as 4.5 to 4. Assuming this rounding mode, the IEEE-754 standard guarantees that the relative error between x and its floating point approximation is always bounded by half of the relative gaps

$$|\frac{x - fl(x)}{x}| \leq \frac{\epsilon_{machine}}{2} =: \mu$$

where μ is often referred to as the unit round-off. Equivalently, we can reformulate the principle to saying there exists some ϵ with $|\epsilon \leq \mu$ such that

$$fl(x) = x(1 + \epsilon).$$

This means that the nearest floating point neighbor of any real number $x \in \mathbb{R}$ which is bounded away from the infinities is exact within a factor $1 + \epsilon$.

1.2 Composition of a floating point number

Let us focus on the internal representation of double precision floating point numbers. Any double-precision floating point number is represented by

$$sign.mantissa.2^{exponent-1023} \tag{2}$$

It uses 64 bits of computer memory, where the sign occupies one bit, the mantissa 52 and the exponent 11 bits. Usually, the sign, mantissa and exponent are given as binary numbers since this is the natural language of a computer. However, sometimes to shorted the notation we prefer decimal numbers. To avoid confusion we use the base as a subindex so that for example

$$11_2 = 3_{10} \;\; and \;\; 11_{10} = 1011_2$$

If the first digit in the mantissa is one, for example

$$mantissa = 1.111010\ldots$$

the floating point number is called normalized. Usually one prefers to change the order of magnitude of the floating point number by adjusting the exponent. However, in order to also be able to represent numbers very close to zero, one allows denormalized floating pointing number, where the mantissa begins with a zero,

$$mantissa = 0.111010\ldots$$

In any case the digit left of the binary point is not stored separately and all 52 bits are used for the fractional part of the mantissa. For this reason it is often referred to as the hidden bit. Since the exponent occupies 11 bits of memory, two of which are reserved, there are a total of $2^{11} - 2 = 2046$ possible values for the exponent. One introduces an offset (or bias) of 1023 to be able to represent small numbers as well. Thus the biased (decimal) range of the exponent is

$$-1022, -1021, \ldots, 0, 1, \ldots, 1023.$$

The computer stores a floating point number in the order sign, exponent and mantissa. So for example $1 = 1 \cdot 20$ is represented by

$$0 \;\; 01111111111 \;\; 0000000000000000000000000000000000000000000000000000_2.$$

The exponent 00000000000_2 is reserved to represent either a signed zero for vanishing mantissa $= 0$ or denormalized numbers for non-vanishing mantissa $\neq 0$. Furthermore, the exponent 11111111111_2 is reserved to represent either the infinities ($\pm$ inf) for mantissa $= 0$ or not a number (nan) for mantissa $\neq 0$. This is quite a useful feature as when writing code

$$exp(1000), exp(-1000), \frac{0}{0}, \frac{0}{\infty}$$

result in

$$\text{inf}, 0, \text{nan}.$$

The first output is an example of overflow, the second one an example for underflow. These exceptions are carefully defined in the IEEE standard. Since the set of floating point numbers is a finite, there exist smallest/largest numbers as well as normalized numbers closest to zero. One can easily verify that these numbers are given by $\pm 1.79769 \cdot 10^{308}$ and $\pm 2^{-1022} \approx \pm 2.22507 \cdot 10^{-308}$.

1.3 The second principle of floating point arithmetic

The beauty of the IEEE-754 standard is that it not just properly defines floating point numbers, rounding modes and exception handling. It also dictates upper error bounds when using basic arithmetic operations on a computer. This is leads us to the second principle of floating point arithmetic. Denote with $\oplus, \ominus, \odot, \oslash$, the computer implementations of addition, subtraction, multiplication and division. It is clear that even if x and y are representable as exact floating point numbers, this is not necessarily true for the result of an arithmetic operation. Take $2^2 + 2^{-52}$, for example, which would be rounded in floating point arithmetic to 4. Hence, the second principle of floating point arithmetic demands that the computer implementations shall give the same result as first adding the two floating point numbers x to y in exact arithmetic and rounding afterwards, i.e.

$$\begin{aligned} x \oplus y = fl(x + y) \quad and \quad x \ominus y = fl(x - y) \\ x \odot y = fl(x \cdot y) \quad and \quad x \oslash y = fl(x/y) \end{aligned}$$

From these properties, we instantly derive the second principle of floating point arithmetic

$$x \otimes y = (x * y)(1 + \delta)$$

for " $\in \{+, -, \cdot, /\}$ and some δ which satisfies $|\delta| \leq \mu$. The second principle of floating point arithmetic shows that the worst relative error that basic operations introduce is again on the order of the unit round-off. This is the best we can hope for. A word of warning: Of course this does not imply that repeated applications of these operations may not accumulate. For example, in Python

$$1e5 * sin(\pi) = 1.2246467991473532e - 11$$

Similarly, it is dangerous to subtract numbers of the same size. The result of the following calculation

$$1.000000000001 - 1 = 1.000088900582341e - 12$$

is accurate only up to five digits. This loss in precision is called cancellation, which is however not the result of an unstable algorithm but of the ill-conditioning of the underlying mathematical problem (in this case the subtraction of numbers which are almost of the same size). It is equally dangerous to add two numbers of greatly varying size as the calculation

$$sqrt(1e - 16 + 1) - 1 = 0$$

demonstrates.

The advantage of the scientific notation(as described in 2) as opposed to fixed-point notation is obvious. Suppose we use a fixed point notation with 5 integer and 6 fractional digits in decimal arithmetic. Then

$$\begin{aligned} \pi &\approx 00003.141592 \\ \frac{\pi}{10000} &\approx 00000.000314 \\ 10000\pi &\approx 31415.926535 \end{aligned}$$

showing that the amount of significant digits depends on the order of magnitude of the number.

The interested reader may find a lot more information on floating point arithmetic in the following books:

- M. Overton. Numerical Computing with IEEE Floating Point Arithmetic. SIAM, 2001.
- N. Higham. Accuracy and Stability of Numerical Algorithms. SIAM, second edition, 2002

Online there are also IEEE calculators which are fun to play around with:

- IEEE-754 Calculator. http://babbage.cs.qc.cuny.edu/IEEE-754/