I. ERRORS

In the last lecture we saw that we will always run into errors while numerically calculating any quantity. These errors stem from the finiteness of the machines to store data. We will talk about the errors a little more detail and point out simple tricks to get around it. We must remember throughout that computer math is very different from the math we have learnt up to now.

Let us start with the following example. Suppose that the problem you need to solve on a machine consists of n steps and let us assume that each step has some probability p of being correct. Then the total probability of getting the right answer after n steps is $P = p^n$. For $p = 0.9993$ and $n = 1000$, we get $P \sim 0.5$. Therefore, the answer you get is as likely to be wrong as right. What we need to do is undertand how to increase the probability of getting the right answer. A lot of times, the errors depend on the parameters we have used. For example, if we need to integrate a function numerically between a and b, then the error depends a lot on the number of points used in the integration, the step size used for the integration grids etc. We need to choose these optimally in order to get reliable results. In the following, we will see lots of examples and ways to get around the blocks!

A. Subtractive Cancellation

Whenever a number is stored in a computer, it is always has a round-off error (unless it is a machine number!) so that a number a becomes:

$$a_c = a(1 + \epsilon_a) \tag{1}$$

Then subtraction of two numbers

$$a = b - c \equiv a_c = b_c - c_c \tag{2}$$

which is

$$\begin{aligned} a_c &= b(1 + \epsilon_b) - c(1 + \epsilon_c) && (3)\\ &= b - c + b\epsilon_b - c\epsilon_c && (4)\\ \Rightarrow \frac{a_c}{a} &= 1 + \frac{b}{a}\epsilon_b - \frac{c}{a}\epsilon_c && (5) \end{aligned}$$

If a is small, then it implies that $b \approx c$. If we subtract two numbers which are very close, we are left with a smaller number, i.e., only the least significant parts where the round-off is maximum. The result of a such a subtraction is very less in significance. That is:

$$\begin{aligned} \frac{a_c}{a} &= 1 + \frac{b}{a}(\epsilon_b - \epsilon_c) && (6)\\ &= 1 + \epsilon_a && (7) \end{aligned}$$

Since a is small, $\frac{b}{a}$ is large and although the relative errors in b and c cancel somewhat, the error in a is augmented by $\frac{b}{a}$. Now if we on the other hand add two numbers, b and c, then the result a is larger than the two numbers. Then $\frac{b}{a}$ is small and the result would be accurate.

Let us consider a series summation for $\exp -x$. For large x, we would end up cancelling large numbers and very soon the series become garbage. What would be a good way to work around this?

B. Multiplicative Errors

What happens when we multiply two number?

$$a = b \times c \Rightarrow a_c = b_c \times c_c \tag{8}$$

$$\Rightarrow \frac{a_c}{a} \approx 1 + \epsilon_b + \epsilon_c \tag{9}$$

Therefore it is possible to be in situations when the error in the final answer due to round-off is larger than the errors in the individual numbers.

We can estimate the round-off error when numbers are stored are manipulated as follows. Let us assume that the computer representation of a number varies randomly from the actual number. Let the error in any number stored be the machine precision ϵ_m. Then after N manipulations, the total error due to round-off becomes,

$$\epsilon_{\rm ro} = \sqrt{N}\epsilon_m \tag{10}$$

similar to a random walk! If a computer can perform 10^{10} operations per second, then in three hours it would have performed $N = 10^{14}$ steps. The error due to round-off would be $\epsilon_{\rm ro} = 10^7 \times \epsilon_m$. Therefore in order to get small errors, $\epsilon_m < 10^{-7}$. This also illustrates how hard it is to get machine precision!

C. Errors in Algorithms

Any problem in Physics is ultimately converted to an algorithm that is solved on a computer. An algorithm is often characterized by a step size h or by the number of steps required to reach the goal (N). Usually one expects that the numerical result approaches the exact one when $h \to 0$ or $N \to \infty$. For any algorithm there is always an *approximation error*, which is the difference between the exact and the numerical result. Given a general algorithm it is important to figure out whether the answer we get is a converged one, how precise are the results and how expensive it is to run. Let us explore a generic way to figure out how the errors propagate.

The total error is a sum of the approximation error, the round-off error, bad data, systematic errors etc. If we arbitrarily increase N, then the computer works really hard, only accumulating errors at each step. It is therefore important to pick out an optimum value for N or h. Let us assume that the total error in some algorithm is dominated by the round-off and the approximation error:

$$\epsilon_{\rm tot} = \epsilon_{\rm ro} + \epsilon_{\rm approx}. \tag{11}$$

We already know that the round-off will be some (positive) power of N, say,

$$\epsilon_{\rm ro} = \sqrt{N}\epsilon_m, \tag{12}$$

for example. The approximation error should decrease with increasing N, one expects an inverse power law such as

$$\epsilon_{\rm approx} = \frac{\alpha}{N^\beta} \tag{13}$$

Therefore the total error is,

$$\epsilon_{\rm tot} = \sqrt{N}\epsilon_m + \frac{\alpha}{N^\beta}. \tag{14}$$

At small values of N one expects that the approximation error would dominate, while at large N values, the round-off error would contribute significantly. Let the exact answer be A. Then the answer $A(N)$ deivates from A as follows:

$$A(N) = A + \frac{\alpha}{N^\beta} \tag{15}$$

for the region where the round-off is not important. Now we can double the number of points N and calculate $A(2N)$. Now their difference $|A(N) - A(2N)|$ on a log-log plot would give a power-law behavior with a negative slope for small N that eventually grows with N. The slope of the small N behavior gives the exponent β. This is an empirical way to figure out the errors and we will see lots of examples.