PHYS 7412
Lecture 1: Errors

- Internal representation of numbers
- Round-off and Truncation errors

No matter how sophisticated modern computers appear to be, at their very essence they are machines that add and subtract numbers.

These numbers are internally represented in a binary (or hexadecimal) format. As physicists, we normally use and think in terms of floating point decimal numbers (perhaps in scientific notation). Most modern computer languages allow us to operate in terms of such numbers, which implies that something non-trivial is already happening between what we “touch and feel” in a computer program and what is internally going on in a computer. This non-trivial interaction is the source of the so called “**round-off**” error.

Moreover, most of physics is done in terms of differential and integral calculus. To do differential and integral calculus we will be forced to represent operations in terms of basic algebraic operations. Again non-trivial things can happen in such a representation. This is the source of the so-called “**truncation**” error.

In most computers numbers are internally stored as a sign bit times a mantissa times an exponential (in base 2 or 16), eating up all the bits used to represent the number,

$$s \times M \times B^{e-E}$$

Where E is a “bias” the machine may use. An example would be for a 32 bit precision, s takes up one bit, the mantissa 23 bits and the exponent 8 bits more.

A given number (say, 10^{-7}) can be represented in multiple ways in such a representation. Most machines “normalize” numbers by “left shifting” them, meaning representing them with the largest possible mantissa and compensating with a negative power of the basis. This allows to have “more significant figures”.

Operations between numbers are not exact, even if they are *individually exactly represented in the machine*.

For instance, to subtract or add two numbers machines “right shift” the mantissa of the smaller number until both numbers have the same exponent. If this procedure requires to right shift past the accuracy of the representation of the mantissa, then the number is treated as zero!

The smallest floating point number that can be added to the number 1.0 producing a result different from 1.0 is called the machine accuracy ε_m. For a typical machine with B=2 and 32 bit accuracy, such number is 10^{-8}.

Almost any arithmetical operation between two numbers in a computer should therefore be assumed to have a round-off error of at least 10^{-8} fractionally, although it is obvious that this could be much larger!

Round-off errors accumulate with successive calculations. If one is lucky and round-off errors accumulate randomly, one gets a fractional error of $\sqrt{N}\varepsilon_m$ if the errors appear randomly.

Unfortunately, this does not have to be the case. In many occasions patterns in the computations or peculiarities of the machine may lead to round-off errors compounding and the final error being $N\varepsilon_m$

In many occasions, round-off errors can compound much more badly, as when one subtracts two numbers that are close to each other, producing a result that is close to the machine precision. A common example of this is the evaluation of the solution of the quadratic equation in the case of close roots, when b^2 is much larger than the product ac.

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

Truncation error, is the error that we make by representing continuous differential and integral calculus via discrete expressions. Although round-off error is a hardware thing that one cannot do much about, truncation error is totally under control of the programmer via the choice of algorithm one makes.

In particular, to help distinguish them, truncation error is the error one gets if one runs one's code on a "perfect" computer that represents numbers to infinite accuracy. Minimizing this error (while guarding for possible interactions with round-off error) is the essence of numerical analysis!

Unfortunately, sometimes both errors interact in non-trivial ways. This is what happens when one has "unstable" algorithms. These are algorithms that would in principle run fine on "perfect" computers but in real ones the round-off error gets horribly compounded in such a way as to render the algorithm unstable. A simple example is to try to compute the powers of the "golden mean",

$$g = \frac{(\sqrt{5}-1)}{2} \approx 0.618034005$$

Using the fact that $g^{n+1}=g^{n-1}-g^{n}$

This might appear as very economical since one only needs to subtract instead of multiply. Unfortunately such algorithm is also solved by

$$g' = -\frac{(\sqrt{5}+1)}{2} \approx -1.11$$

Since it is a linear relation and the second solution is greater than one, any initial contamination with the wrong solution explodes exponentially.

This is an example of what is called a “stiff” system (this terminology is used in linear differential equations) in which there are two solutions to a problem, one much larger in general than the other. Therefore any contamination in the initial data with the “large” solution quickly “swamps” the smaller one due to round-off error, even if the discretization algorithm used would work in a “perfect” computer.

Handling round-off errors is largely a “craft” or “art” that can only be learned effectively through experience, since its occurrence, and the possible cures are on a case-by-case basis. In this course we will therefore make here a big point about it initially so we are all aware of it, but will from here on assume that round-off errors are not existent and concentrate on handling truncation errors.