CS412: Lecture #1

Mridul Aanjaneya

January 20, 2015

Types of Errors

1. When doing integer calculations one can many times proceed exactly, except of course in certain situations, e.g. division 5/2=2.5. However, when doing floating point calculations, rounding errors are the norm, e.g. 1./3. = .3333333... cannot be expressed on the computer. Thus, the computer commits rounding errors to express numbers with machine precision, e.g. 1./3. = .3333333. Machine precision is 10^{-7} for single precision and 10^{-16} for double precision. Rounding errors are only one source of approximation error when considering floating point calculations. Some others are listed below.

2. Approximation errors come in many forms:

   (a) **empirical constants:** Some numbers are unknown and measured in a laboratory only to limited precision. Others may be known more accurately but limited precision hinders the ability to express the numbers on a finite precision computer. Examples include Avogadro's number, the speed of light in vacuum, the charge on an electron, Planck's constant, Boltzmann's constant, pi, etc. Note that the speed of light is 299792458 m/s *exactly*, so we are ok for double precision but not single precision.

   (b) **modeling errors:** Parts of the problem under consideration may simply be ignored. For example, when simulating solids or fluids, sometimes frictional or viscous effects respectively are not included.

   (c) **truncation errors:** These are also sometimes called discretization errors and occur in the mathematical approximation of an equation as opposed to the mathematical approximation of the physics (i.e., as in modeling errors). We will see later that one cannot take a derivative or integral exactly on the computer so we approximate these with some formula (recall Simpson's rule from your Calculus class).

   (d) **inaccurate inputs:** Many times we are only concerned with part of a calculation and we receive a set of input numbers and produce a

set of output numbers. It is important to realize that the inputs may have been previously subjected to any of the errors listed above and thus, may already have limited accuracy. This can have implications for algorithms as well, e.g., if the inputs are only accurate to 4 decimal places, it makes little sense to carry out the algorithm to an accuracy of 8 decimal places. This issue commonly resurfaces in scientific visualization or physical simulation where experimental engineers can be unhappy with visualization algorithms that are "lossy", meanwhile forgetting that the part that is lost may contain no useful, or accurate information whatsoever.

3. In dealing with errors, we will refer to both the absolute error and the relative error.
   (a) **Absolute error** = |approximate value - true value|
   (b) **Relative error** = absolute error/|true value|

Our motivation for studying numerical analysis and scientific computing

In many instances we are faced with mathematical problems where the conventional method "on paper" does not perform as well, when implemented on the computer. Take, for example, the quadratic equation

$$ax^2 + bx + c = 0$$

In theory, we have a perfectly conclusive method for determining the roots of this equation; simply use the quadratic formula

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{1}$$

In fact, let us consider a very specific example:

$$0.0501x^2 - 98.78x + 5.015 = 0$$

The actual roots of this equation, rounded to 10 digits of accuracy, are:

$$x_1 = 1971.605916, \qquad x_2 = 0.05077069387$$

We then proceed to evaluate the formula (1) on a computer. Let us assume that the computer stores all intermediate results of this computation with 4 significant digits. Thus, we obtain the following answers:

$$x_1 = 1972, \qquad x_2 = 0.0998$$

Although the value of x_1 is reasonably accurate (it is, in fact, the correct answer to 4 significant digits), the value x_2 is off by almost 100% from the intended

value even when rounded to 4 digits. The reason for this behavior is hiding in the inaccurate (approximate) representation of the intermediate steps of this calculation. Particularly, the computer algorithm approximates $\sqrt{b^2 - 4ac} \approx 98.77$, discarding any subsequent digits due to its limited available precision. Substituting this value into the quadratic formula yields:

$$x_{1,2} = \frac{98.78 \pm 98.77}{0.1002}$$

The numerator involves either a sum or a difference of two near-equal quantities. When adding the 2 values, we compute the reasonably accurate $x_1 = 1972$. When subtracting, all the information that was discarded by truncating $\sqrt{b^2 - 4ac} \approx 98.77$ is now dominating the computed value of the numerator. As a consequence, we obtain the compromised value $x_2 = 0.0998$. In general, subtraction *reveals* the error, and division by a small number *amplifies* it.

So, subtracting two near-equal quantities is risky ... can we avoid doing so? There is, in fact, an alternative formulation:

$$\begin{aligned} x_{1,2} &= \frac{(-b \pm \sqrt{b^2 - 4ac})(-b \mp \sqrt{b^2 - 4ac})}{2a(-b \mp \sqrt{b^2 - 4ac})} \\ &= \frac{b^2 - (b^2 - 4ac)}{2a(-b \mp \sqrt{b^2 - 4ac})} = \frac{4ac}{2a(-b \mp \sqrt{b^2 - 4ac})} \\ &= \frac{2c}{-b \mp \sqrt{b^2 - 4ac}} \end{aligned} \tag{2}$$

Equation (2) subtracts 2 quantities in the denominator where (1) adds them and vice-versa. This technique is called *derationalization*. Applying this approach, we obtain:

$$x_1 = 1003, \qquad x_2 = 0.05077$$

Here the roles are reversed, x_2 is quite accurate, but x_1 is off by about 50% from the intended value. One might suggest that we selectively use (1) or (2) to compute x_1 or x_2 respectively, and avoid the subtraction-induced inaccuracies. Although this may be realistic in this isolated example, in more complex problems that arise in practice, performing such a case-by-case handling may prove impractical. In many cases, it may even be impossible to predict that a value lacks accuracy (not knowing the exact value beforehand).

Let us take a brief detour and look at an iterative scheme for computing $\sqrt{a}$. Starting with an initial guess x_0, we iterate as follows:

$$x_{k+1} = \frac{x_k^2 + a}{2x_k}$$

Assuming this scheme converges, i.e., $\lim_{k\to\infty} x_k = A$, gives

$$A = \frac{A^2 + a}{2A} \Rightarrow 2A^2 = A^2 + a \Rightarrow A^2 = a \Rightarrow A = \sqrt{a}$$

Note how we computed $\sqrt{a}$ with just addition, multiplication and division operations! The same principle can be applied to solve a quadratic equation as well. Consider the following scheme (which is a special case of what we will later describe as Newton's method):

- Start with an initial guess x_0 of the solution.
- Iterate

$$x_{k+1} = \frac{ax_k^2 - c}{2ax_k + b}$$

- After N iterations, take x_N as the estimated solution.

Let us try it ... start with a guess $x_0 = 1$

$$x_0 = 1, \quad x_1 = 0.050313\ldots, \quad x_2 = 0.0507706\ldots$$

After just 2 iterations, x_2 is correct to more than 6 significant digits! Let us aim for the other solution by setting $x_0 = 2000$

$$x_0 = 2000, \quad x_1 = 1972.003\ldots, \quad x_2 = 1971.60599\ldots$$

As is typically the case, there are trade-offs to consider: with this iterative method, we require an "initial guess", and there is little guidance offered on how to pick a good one. Also, we need a few iterations before we can obtain an excellent approximation. On the other hand, the method does not require any square roots, thus being significantly cheaper in terms of computation cost per iteration. In general, the two different methodologies that we have seen fall into two broad classes:

- **Direct methods:** These methods give a "recipe" for directly obtaining the final solution. The closed-form formulas we saw above for the roots of a quadratic equation fall in this category.
- **Iterative methods:** These methods obtain better approximations to the solution through successive iterations.

The traits and trade-offs of such methods will be a point of focus for CS412, and are central to the field we call numerical analysis and scientific computing.