LECTURE 3

Computer Arithmetic

Anyone who has fiddled with a calculator long enough knows that it's relatively easy to trick it into producing ridiculous answers. Indeed, in spite of the possibility of producing answers with a huge number of decimal places, it is important to understand right from the start that (non-trivial) numerical calculations are always wrong. And so unless one knows how wrong a numerical calculation might be, it is hard to say how accurate the corresponding result might be. In order to determine the size of the error in a numerical calculation, it is crucial to first understand how computers calculate, how calculational errors are introduced, and how errors propagate.

1. Floating Point Numbers and Machine Numbers

The way computers handle integers is very straight-forward and well-known: an integer is represented simply by a string of 0's and 1's corresponding to the number's representation in base 2. However, in scientific analysis, one deals more commonly with the decimal approximations of real numbers. Thus, π is represented as

$$\pi = 3.1459265359\ldots$$

Computers represent real numbers in terms of a data type known as a *floating point number*. While the precise implementation of floating point numbers depending on the binary *word length* (which depends upon the architecture and software) of the computer. On a 32-bit system it works like this. A binary floating point number is parameterized by three binary integers:

$$x = (-1)^s(1.f)2^{e-127}$$

where s is either 0 or 1, f is a (23-bit) binary integer between 0 and $2^{23}-1$ and e is an (eight-bit) binary integer between 0 and 2^8-1. Thus, for example

$$\begin{aligned}(-4.125)_{10} &= -\left(2^2+2^{-3}\right)\\ &= (-1)^1(1.00001)_2 2^{129-127}\\ &= (-1)^1(1.00001)_2 2^{(010000001-00111111)_2}\end{aligned}$$

so the binary floating point representation of -4.125 on a 32-bit computer would be the binary string

1 01000000 00001000000000000000000

Now, of course, not every real number can be represented in this form. Not only do we have a limitation in precision (due to the fact that we allow only 23 bits for precribing decimal places) but since the exponent total exponent $e-127$ must lie between -126 and $+127$ (the extreme cases when $e=0$ or 255 are reserved for handling 0, infinity and "not a number" errors).

On a different architecture (e.g. a 64-bit machine) the number of binary digits reserved for the exponent e and the mantisa $e-127$ will be different, and so so will the range of numbers that can be represented in this fashion. To make this dependence on machine architecture manifest, we usually refer to the binary floating point representation of a real number x as the **machine number** corresponding to x.

2. Rounding Errors

We shall now analyze the error introduced in approximating a given real number x by a machine number $fl(x)$. Let us assume that we're working on a machine that reserves p bits for the prescription of the mantissa portion of a binary floating point number. We also assume that

$$x = q \times 2^m$$

where q is a real number greater than or equal to 1 but less then 2, and m is an integer exponent within the bounds of the machine (except for excluding astronomically huge and infinisimally small numbers, we haven't yet made any assumptions about the real number x). Let

$$q = 1.a_1a_2 \dots a_p a_{p+1} \dots$$

be the binary decimal expansion of q. Now in the machine representation of q we must truncate this decimal expansion after the p^{th} term. One way to do this is to simply *round down* by simply chopping off every digits after the p^{th} one. Set

$$q_- = 1.a_1a_2 \dots a_p$$

Another way to do this is to round up, by adding 2^{-p} to q_-:

$$\begin{array}{rcccccccc} q_+ & = & 1.a_1a_2 \dots a_p + 2^{-p} & & & & & & \\ & = & 1 & . & a_1 & \dots & a_{p-1} & a_p \\ & + & 0 & . & 0 & \dots & 0 & 1 \end{array}$$

(effectivly increasing the last digit of q_- by 1). Clearly,

$$q_- \le q \le q_+$$

and

$$|q - q_\pm| \le |q_+ - q_-| = 2^{-p}$$

Actually, if the machine is smart it will also be able to tell which machine mantissa q_- or q_+ is actually closer to x. Let us denote by q^* the machine number closest to q. In this case,

$$|q - q^*| \le \frac{1}{2} |q_+ - q_-| = 2^{-p-1}$$

and so the **absolute error** in replacing x by the machine number closest to it is

$$|q \times 2^m - q^* \times 2^m| = |q - q^*| \times 2^m \le 2^{m-p-1}$$

and the corresponding **relative error** is

$$\frac{|q \times 2^m - q^* \times 2^m|}{|q \times 2^m|} \le \frac{1}{q} 2^{-p-1} \le 2^{-p-1}$$

In summary, if x is a real number and $fl(x)$ is the machine number closest to x then

$$\left| \frac{x - fl(x)}{x} \right| \le 2^{-p-1}$$

Denoting the relative error $(x - fl(x))/x$ by δ, we can represent this inequality by the following formula

$$fl(x) = x(1 + \delta) \quad , \quad |\delta| \le 2^{-p-1}$$

For a machine that uses p binary digits to represent the mantissa of a machine number, the number $\varepsilon = 2^{-p-1}$ that bounds the relative error in converting real numbers to machine numbers is called the **unit roundoff error** for the computer.

EXAMPLE 3.1. Suppose a hypothetical computer uses 8 binary digits to represent the mantissa of a floating point number. Find the machine number closest to 1.10.

- Using binary long division one can verify that

$$1.10 = 1 + \frac{1}{10} = (1.00011001100110011\ldots)_2$$

Thus, in the floating point representation on the given machine the mantisa is either

$$q_- = (1.00011001)_2 = (1.09765625000\ldots)_{10}$$

or

$$q_+ = q_- + (0.00000001)_2 = (1.00011010)_2 = (1.1015625\ldots)_{10}$$

We have

$$\begin{aligned} |q - q_-| &= (1.1)_{10} - (1.09765625)_{10} = (0.00234375000\ldots)_{10} \\ |q - q_+| &= (1.1015625)_{10} - (1.1)_{10} = (0.001562500\ldots)_{10} \end{aligned}$$

Since q_+ is closest to q we have

$$fl(1.1) = q_+ = (1.000110001)_2$$

3. Floating Point Error Analysis

Let us now explore how rounding errors are propagated in calculations. We shall assume that when a machine calculates the sum, difference, product, or quotient of two numbers, that it first carries out the operation with absolute precision, then converts the result to an infinite precision floating point representation and finally rounds the answer off. Hence, letting $\odot$ denote any of the arithmetic operators $+, -, *$, or $\div$, we have

$$fl(x \odot y) = (x \odot y)(1 + \delta) \quad , \quad |\delta| \leq \varepsilon$$

where $\varepsilon = 2^{-p-1}$ is the relative error for computer. In the formula above, we assume that x and y are known to the computer as machine numbers (as might occur an intermediate stage of a calculation). Of course, if we start out with real numbers x and y, then they must first be converted into machine numbers. and so the result of a machine computation would be

$$\begin{aligned} fl\left(fl(x) \odot fl(y)\right) &= \left(fl(x) \odot fl(y)\right)(1 + \delta_1) \quad , \quad |\delta_1| \leq \varepsilon \\ &= \left(x(1 + \delta_2) \odot y(1 + \delta_3)\right)(\bar{1} + \delta_1) \quad , \quad |\delta_2| . |\delta_3| \leq \varepsilon \end{aligned}$$

Example 3.2. Suppose x, y, z are machine numbers on a computer with unit roundoff error 2^{-p-1}. Describe the error inherent in computing $x(y + z)$.

-

$$\begin{aligned} fl\left(x(y+z)\right) &= (x * fl(y+z)(1+\delta_1)) \quad , \quad |\delta_1| \leq 2^{-p-1} \\ &= fl\left[x * fl(y+z)\right](1+\delta_2)(1+\delta_1) \quad , \quad |\delta_1|, |\delta_2| \leq 2^{-p-1} \\ &= fl\left(x * fl(y+z)\right)(1 + \delta_1 + \delta_2 + \delta_1\delta_2) \quad , \quad |\delta_1|, |\delta_2| \leq 2^{-p-1} \\ &\approx fl\left(x * fl(y+z)\right)(1+\delta_3) \quad , \quad |\delta_3| \leq 2^{-p} \end{aligned}$$

In the first step, $y + z$ is computed and then converted into a machine number. The corresponding relative error in rounding off $y + z$ is signified by δ_1. Then the machine number x is multiplied by the machine number $fl(z + z)$ with corresponding relative error δ_2. Finally, the effective relative error is consolidated in a single term $\delta_3 = \delta_1 + \delta_2 + \delta_1\delta_2 \approx \mathcal{O}\left(2^{-p}\right)$.