

Thursday January 26th 2012

How are numbers stored on the computer?

First, we shall review the concept of “scientific notation”, which will give us some helpful insights. For any decimal number x (we assume that x is a terminating decimal number, with finite nonzero digits) we can write

$$x = a \times 10^b, \text{ where } 1 \leq |a| < 10$$

Exception: When $x = 0$, we simply set $a = b = 0$. For example:

| x (decimal notation) | x (scientific notation) |
|------------------------|---------------------------|
| 2012 | 2.012×10^3 |
| 412 | 4.12×10^2 |
| 3.14 | 3.14×10^0 |
| 0.000789 | 7.89×10^{-4} |
| 0.2091 | 2.091×10^{-1} |

Corresponding
textbook
chapter(s):
§1.2

Every decimal (or Base-10) number can be written

$$a_k a_{k-1} \cdots a_2 a_1 a_0 . a_{-1} a_{-2} a_{-3} \cdots a_{-l} = \sum_{i=-l}^{+k} a_i 10^i$$

For example

| x | a_3 | a_2 | a_1 | a_0 | a_{-1} | a_{-2} | a_{-3} |
|-------|-------|-------|-------|-------|----------|----------|----------|
| 3.14 | | | | 3 | 1 | 4 | |
| 0.037 | | | | | | 3 | 7 |
| 2012 | | 2 | 0 | 1 | 2 | | |

Binary (Base-2) fractional numbers are written

$$b_k b_{k-1} \cdots b_2 b_1 b_0 . a_{-1} b_{-2} b_{-3} \cdots b_{-l} (2) = \sum_{i=-l}^{+k} b_i 2^i$$

where every digit b_i is now only allowed to equal 0 or 1. For example

- $5.75 = 4 + 1 + 0.5 + 0.25 = 1 \times 2^2 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = 101.11_{(2)}$
- $17.5 = 16 + 1 + 0.5 = 1 \times 2^4 + 1 \times 2^0 + 1 \times 2^{-1} = 10001.1_{(2)}$
- $5.75 = 4 + 1 + 0.5 + 0.25 = 1 \times 2^2 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = 101.11_{(2)}$

Note that certain numbers are finite (terminating) decimals, actually are periodic in binary, e.g.

$$0.4_{(10)} = 0.01100110011\dots_{(2)} = 0.0011\overline{0011}_{(2)}$$

Machine numbers (a.k.a. binary floating point numbers)

The numbers stored on the computer are, essentially, “binary numbers” in scientific notation $x = \pm a \times 2^b$. Here, a is called the *mantissa* and b the *exponent*. We also follow the convention that $1 \leq a < 2$; the idea is that, for any number x , we can always divide it by an appropriate power of 2, such that the result will be within $[1, 2)$. For example:

$$x = 5_{(10)} = 1.25_{(10)} \times 2^2 = 1.01_{(2)} \times 2^2$$

Thus, a machine number is stored as:

$$x = \pm 1.a_1a_2\cdots a_{k-1}a_k \times 2^b$$

- In *single precision* we store $k = 23$ binary digits, and the exponent b ranges between $-126 \leq b \leq 127$. The largest number we can thus represent is $(2 - 2^{-23}) \times 2^{127} \approx 3.4 \times 10^{38}$.
- In *double precision* we store $k = 52$ binary digits, and the exponent b ranges between $-1022 \leq b \leq 1023$. The largest number we can thus represent is $(2 - 2^{-52}) \times 2^{1023} \approx 1.8 \times 10^{308}$.

In other words, single precision provides 23 binary significant digits; in order to translate it to familiar decimal terms we note that $2^{10} \approx 10^3$, thus 10 binary significant digits are roughly equivalent to 3 decimal significant digits. Using this, we can say that single precision provides approximately 7 decimal significant digits, while double precision offers slightly more than 15.

Absolute and relative error

All computations on a computer are approximate by nature, due to the limited precision on the computer. As a consequence we have to tolerate some amount of *error* in our computation. Actually, the limited machine precision is only one source of error – other factors may further compromise the accuracy of our computation (in later lectures we will discuss *modeling*, *truncation*, *measuremenet* and *roundoff* errors). At any rate, in order to better understand errors in computation, we define two error measures: The absolute, and the relative error. For both definitions, we denote by q the exact (analytic) quantity that we expect out of a given computation, and by \hat{q} the (likely compromised) value actually generated by the computer.

Absolute error is defined as $e = |q - \hat{q}|$. This is useful when we want to frame the result within a certain interval, since $e \leq \delta$ implies $q \in [\hat{q} - \delta, \hat{q} + \delta]$.

Relative error is defined as $e = |q - \hat{q}|/|q|$. The result may be expressed as a percentile and is useful when we want to assess the error relative to the value of the exact quantity. For example, an absolute value of 10^{-3} may be insignificant when the intended value q is in the order of 10^6 , but would be very severe if $q \approx 10^{-2}$.

Rounding, truncation and machine ϵ (epsilon)

When storing a number on the computer, if the number happens to contain more digits than it is possible to represent via a machine number, an approximation is made via *rounding* or *truncation*. When using truncated results, the machine number is constructed by simply discarding significant digits that cannot be stored; rounding approximates a quantity with the *closest* machine-precision number. For example, when approximating $\pi = 3.14159265\dots$ to 5 decimal significant digits, truncation would give $\pi \approx 3.15159$ while the rounded result would be $\pi \approx 3.1516$. Rounding and truncation are similarly defined for binary numbers, for example $x = 0.1011011101110_{(2)}\dots$ would be approximated to 5 binary significant digits as $x \approx 0.1011_{(2)}$ using truncation, and $x \approx 0.10111_{(2)}$ when rounded.

A concept that is useful in quantifying the error caused by rounding or truncation is the notion of the machine ϵ (epsilon). There are a number of (slightly different) definitions in the literature, depending on whether truncation or rounding is used, specific rounding rules, etc. Here, we will define the machine ϵ as the smallest positive machine number, such that

$$1 + \epsilon \neq 1 \quad (\text{on the computer})$$

Why isn't the above inequality always true, for any $\epsilon > 0$? The reason is that, when subject to the computer precision limitations, some numbers are "too small" to affect the result of an operation, e.g.

$$\begin{aligned} 1 &= 1.\underbrace{000\dots000}_{23 \text{ digits}}_{(2)} \times 2^0 \\ 2^{-25} &= 0.\underbrace{000\dots000}_{23 \text{ digits}}01_{(2)} \times 2^0 \\ 1 + 2^{-25} &= 1.\underbrace{000\dots000}_{23 \text{ digits}}01_{(2)} \times 2^0 \end{aligned}$$

When rounding (or truncating) the last number to 23 binary significant digits corresponding to single precision, the result would be exactly the same as the

representation of the number $x = 1$! Thus, on the computer we have, in fact, $1 + 2^{-25} = 1$, and consequently 2^{-25} is smaller than the machine epsilon. We can see that the smallest positive number that would actually achieve $1 + \epsilon \neq 1$ with single precision machine numbers is $\epsilon = 2^{-24}$ (and we are even relying a “round upwards” convention for tie breaking to come up with a value this small), which will be called the machine ϵ in this case. For double precision the machine ϵ is 2^{-53} .

The significance of the machine ϵ is that it provides an upper bound for the relative error of representing any number to the precision available on the computer; thus, if $q > 0$ is the intended numerical quantity, and \hat{q} is the closest machine-precision approximation, then

$$(1 - \epsilon)q \leq \hat{q} \leq (1 + \epsilon)q$$

where ϵ is the machine epsilon for the degree of precision used; a similar expression holds for $q < 0$.