Numerical Error Analysis

Analytic representation of numbers

1) Integers : $0, \pm 1, \pm 2, \pm 3, \ldots$. Let x be an integer.

$$\boxed{x = (\alpha_0 + \alpha_1 b + \alpha_2 b^2 + \cdots + \alpha_n b^n)\, s}$$ where

a) b natural number > 1 $(2, 3, 4, \ldots)$. $\longleftrightarrow$ base of representation

b) For all κ : $0 \leq \alpha_\kappa < b$, and natural $\longleftrightarrow$ digits

c) $s = +1$ or $s = -1$ $\longleftrightarrow$ sign.

example

$666 = 6 + 6 \cdot 10 + 6 \cdot 100$ (decimal)

$666 = 0 + 1 \cdot 2 + 0 \cdot 4 + 1 \cdot 8 + 1 \cdot 16 + 0 \cdot 32 + 0 \cdot 64 + 1 \cdot 128 + 0 \cdot 256 + 1 \cdot 512 = (1010011010)_2$ (Binary)

$666 = 10 + 9 \cdot 16 + 2 \cdot 16^2 = (29A)_{16}$ (hexadecimal).

↳ use $A, B, C, D, E, F = 10, 11, 12, \ldots, 15$.

▶ note the relation between binary and hexadecimal:

$(1010011010)_2$

$(2 \quad 9 \quad A)_{16}$

$(1010)_2 = 10 = (A)_{16}$

$(1001)_2 = 9 =$

$(10)_2 = 2 =$

Every 4 digits of the binary representation correspond to 1 digit of hexadecimal. This can be used in binary $\longleftrightarrow$ hexade conversion. ◀

▶ integer division : $\boxed{\frac{\alpha}{b} = q + \frac{r}{b}}$ q = quotient, r = remainder $< b$

$q = a/b$ ← integer division

$r = a\%b$ ← remainder operation.

] convention in C and C++ ◀

Converting to decimal is easy. Converting from decimal as follows

Algorithm:

Let $x>0$ be a given integer
$n = \max\{k \in \mathbb{N} \mid b^k < x\}$
$x_n = x$
for $k = n, n-1, \dots, 1, 0$
  $a_k = x_k / b^k$
  $x_{k-1} = x_k \,\%\, b^k$, if $k>0$

2) **Floating point numbers**: 3.45, π, $\sqrt{2}$

$$x = s\, b^e \left[\frac{m_1}{b} + \frac{m_2}{b^2} + \frac{m_3}{b^3} + \dots \right]$$

a) b natural number >1 $(2,3,4,\dots)$ $\leftrightarrow$ base
b) e integer $(0, \pm 1, \pm 2, \dots)$ $\leftrightarrow$ exponent
c) $s=1$ or $s=-1$ $\leftrightarrow$ sign
d) For all k: $m_k \in \mathbb{N}$ and $0 \le m_k < b$ $\leftrightarrow$ mantissa

To convert to base b:

Algorithm:

$e = \min\{k \in \mathbb{Z} \mid b^k > x\}$
$x_0 = x / b^e$
for $k = 0, 1, 2, \dots$
  $m_k = \lfloor x_k b \rfloor$
  $x_{k+1} = b x_k - \lfloor b x_k \rfloor$

▶ Note that if $x_j = x_k$ with $j<k$ then, the sequence (m_k) will repeat after the k. ◀

▶ If $x_k = 0$ for some k, then the algorithm may terminate. Then we say that x is terminating with respect to b. ◀

Numerical representation of numbers.

1) Short integers (16-bit) $\longrightarrow x = s(a_0 + a_1 2 + \cdots + a_{14} 2^{14})$
2) Integers (32 bit) $\longrightarrow x = s(a_0 + a_1 2 + \cdots + a_{30} \cdot 2^{30})$
3) Long integers (64 bit) $\longrightarrow x = s(a_0 + a_1 2 + \cdots + a_{62} \cdot 2^{62})$
4) Unsigned short integer $\longrightarrow x = a_0 + a_1 2 + \cdots + a_{15} \cdot 2^{15}$
5) Unsigned integer $\longrightarrow x = a_0 + a_1 2 + \cdots + a_{31} \cdot 2^{31}$
6) Unsigned long integer $\longrightarrow x = a_0 + a_1 2 + \cdots + a_{63} \cdot 2^{63}$

with $s = 1, -1$ and $a_k = 0, 1$

▶ Integer arithmetic is accurate except for <u>overflow</u> and <u>underflow</u> errors. ◀

▶ In general: k-bit integer $\longrightarrow x = s(a_0 + a_1 2 + \cdots + a_{k-2} 2^{k-2})$
k-bit unsigned integer $\longrightarrow x = a_0 + a_1 2 + \cdots + a_{k-1} 2^{k-1}$

It is also possible to implicitly modify the range by assuming that x is shifted by a given amount. ◀

Numerical floating point representation of numbers.

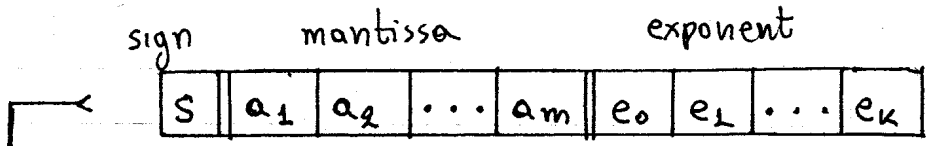
There are many variants. We discuss the IEEE standard.
In general:

$$x = s\, b^e \left[\frac{a_1}{b} + \frac{a_2}{b^2} + \cdots + \frac{a_m}{b^m} \right]$$

$$e = \mu + e_0 + e_1 \cdot b + \cdots + e_k \cdot b^k$$

where, b = base
m = mantissa length
a_k = mantissa bits.
e_k = exponent bits
e = exponent
k = exponent bitlength.
μ = smallest exponent
$M = \mu + (b-1) + (b-1)b + \cdots + (b-1)b^k$
= biggest exponent
s = sign bit.

The actual information that needs to be stored is:



$2 \cdot 2^m \cdot 2^{k+1}$ possible numbers can be represented when $b = 2$.

Rounding function and floating point operations.

Let $\mathcal{F}$ be the set of all the numbers that can be generated by a floating point representation.

A floating point system consists of

1. A rounding function $rd : \mathbb{R} \to \mathcal{F}$ that maps every real number to a representable number.
2. The floating point operations that map a pair of representable numbers to another representable number.

$$x \oplus y \equiv rd(x+y) = \overline{x+y}$$
$$x \ominus y \equiv rd(x-y) = \overline{x-y}$$
$$x \odot y \equiv rd(xy) = \overline{xy}$$
$$x \oslash y \equiv rd(x/y) = \overline{x/y}$$

← book notation

The computer should perform these operations in a manner that is always consistent with the rounding function, for all $x, y \in \mathcal{F}$. (pairs of representable numbers).

Machine epsilon

Definition: The machine epsilon *eps* of a floating point system is the largest number such that:

$$1 \oplus eps = 1$$

This is **NOT** the smallest number that can be represented in the floating point system. It instead measures the ability of the floating point system to handle numbers with very big differences in the order of magnitude.

The machine epsilon provides an estimate of the relative error that is introduced during a fundamental floating point operation:

$$(x \oplus y) = (x+y)(1+\varepsilon_1), \quad |\varepsilon_1| < eps$$
$$(x \odot y) = (xy)(1+\varepsilon_2), \quad |\varepsilon_2| < eps$$
$$(x \ominus y) = (x-y)(1+\varepsilon_3), \quad |\varepsilon_3| < eps$$
$$(x \oslash y) = (x/y)(1+\varepsilon_4), \quad |\varepsilon_4| < eps$$

Note that ε_1, etc are dependent on the specific $x, y \in \mathbb{F}$, nevertheless are always bound by eps.

A note on error propagation

The complete theory of error propagation is discussed in J. Stoer, R. Bulirsch, "Introduction to Numerical Analysis" Floating point operations are fundamentally different from the corresponding analytic ones. For example; if:

$a = 0.23371258 e \bar{+}04$

$b = 0.33678429 e+02$

$c = -0.33677811 e+02$

then

$a \oplus (b \oplus c) = 0.64137126 e-03$ accurate

$(a \oplus b) \oplus c = 0.64100000 e-03$ very inaccurate.

The order with which operations are performed can make a very big difference. There is a formal theory, due to Bauer, with which you can prove whether a computation scheme is or isn't accurate. We will not cover this theory in this course. We will only state a prediction:

▶ Let $y = \varphi(\vec{x})$ be a function of many variables. The inherent error is defined as:

$$\Delta^{(0)} y = [\, |\nabla \varphi(\vec{x}) \cdot \vec{x}| + |y| \,] eps$$

If a computation scheme is "accurate", it should exhibit an

error that has the same order of magnitude as $\Delta^\circ y$ ◀
Therefore, if you observe an error $\gg \Delta^\circ y$, then you have a bug.
▶ Problems with steep gradient $\nabla\varphi(\vec{x})$ have a large inherent error, and it is impossible to eliminate such error.
We say that such problems are <u>ill-conditioned.</u> ◀
The following rules of thumb help.
1) Add large sequences of numbers in <u>ascending</u> order.
2) Avoid $x-y$ if $x-y \ll x$ or $x-y \ll y$.
3) Avoid x/y if $y \ll 1$
4) Avoid $x+y$ if $y \ll x$ or $x \ll y$. (e.g. $1+\text{eps} = 1$).
5) Avoid $\sin x, \cos x, \tan x$ for $x \gg 2\pi$
6) Use Horner scheme to evaluate polynomials.
7) Be paranoid - do formal analysis and proof if your computation is responsible for smart missilles and stuff like that, otherwise you might piss off China.

<u>Rounding functions.</u>

Rounding is implicit in the implementation of floating point arithmetic; there should exist a rounding function such that the floating point operations are consistent with it. How that is actually done is messy bussiness. Computers now do it in hardware not software.

Note that although computers grok binary numbers, they may choose to round in a different system, like <u>octal</u> or <u>hexadecimal</u>, instead of binary. I.e. they carry out operations in binary such that rounding in another system is satisfied.

There is two basic types of rounding.
1) <u>Chopping</u>: Simply drop the digits that can not be resolved.
May cause rounding error up to eps.

2) Symmetric chopping: Drop the digits but round up or down, whichever is nearest.

e.g. $rd(3.14159) = 3.1416$

$rd(3.422) = 3.42$

In case of a tie, pick the even number (i.e. choose "randomly" to minimize statistical bias)

e.g $rd(3.135) = 3.14$

$rd(3.125) = 3.12$

Other conventions have been to round towards $0, +\infty$, or $-\infty$.

↳ Repeating numbers.

Fact: $\boxed{0 \leq |a| < 1 \Rightarrow 1 + a + a^2 + \cdots = \frac{1}{1-a}}$ (geometric series theorem).

Analytically $1/5 = 0.2$ seems a trivial computation. However, in binary arithmetic $1/5$ is a repeating number so it is rounded, and upon conversion to decimal, you might get 0.1999999 or 0.2000001. To see that $1/5$ is repeating:

$$\frac{1}{5} = \frac{1}{4-1} = \frac{1}{4}\frac{1}{1-1/4} = \frac{1}{4}\left[1 + \frac{1}{4} + \frac{1}{4^2} + \cdots\right] =$$

$$= \sum_{k=1}^{+\infty} \frac{1}{4^k} = \sum_{k=1}^{+\infty} \frac{1}{2^{2k}} = 0.01010101\cdots = 0.\overline{01}$$

So repetitiveness depends on the representation system. In general, the following theorem can be proven:

Theorem: Let $0 < B_1 < B_2$ integers such that $B_1 | B_2$ (B_1 divides B_2). Then the following are true:

a) If x repeats in $B_2 \Rightarrow x$ repeats in B_1

b) There exists an x that does not repeat in B_2 but repeats in B_1.