IEEE-754 Arithmetic, binary, and all that.

We have now discussed floating point and the propagation of roundoff error in a very detailed way. Our last job is to see how theory meets practice inside a computer.

Bases and binary

Recall

734.25 means $7 \times 100 + 3 \times 10 + 4 \times 1 + 2 \times \frac{1}{10} + 5 \times \frac{1}{100}$

or

$734.25 = 7 \times 10^2 + 3 \times 10^1 + 4 \times 10^0 + 2 \times 10^{-1} + 5 \times 10^{-2}$

decimal point here

Definition. A binary number is a collection of digits

$$\ldots\ldots d_2 d_1 d_0 . d_{-1} d_{-2} d_{-3} \ldots\ldots = b$$

where each d_i is 1 or 0 and

$$b = \sum_{i=-\infty}^{\infty} d_i \times 2^i$$

~~If all d_i with $i < -K$ are 0, then b is rational and~~

Examples.

$$101 = 1\times 2^2 + 0\times 2^1 + 1\times 2^0 = 5$$

$$0.11 = 1\times 2^{-1} + 1\times 2^{-2} = \frac{1}{2} + \frac{1}{4} = 3/4$$

$$0.\overline{1} = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots\ldots = 1.$$

~~We define~~

Definition. Binary floating point arithmetic is defined by a number of digits n and a range of exponents $m \le e \le M$. The ~~representable~~ in-range numbers are in the form

$$\pm d_0 . d_1 \dots d_n \times 2^e$$

where each d_i is 0 or 1 and d_0 is not zero.

Note: This means that $d_0 = 1$, so we don't have to store it! So we can write the number as

$$\pm 1 . d_1 \dots d_n \times 2^e$$

Example. Suppose $-1022 \le e \le 1023$. Then the smallest positive number is

$$+1.0 \dots 0 \times 2^{-1022} \approx 10^{-307.653}$$

and the largest positive number is

$$+1.1\ldots.1\times 10^{1023}\approx 10^{308}$$

~~Example. If n = 53, then~~

Fact. In n-digit binary floating point

$$\frac{|x-fl(x)|}{|x|}\leq 2^{-n-1}$$

for all in-range numbers x.

Note. Two things are slightly different – this is "half the value of the last digit" as in decimal n-digit floating point, so it's $\frac{1}{2}2^{(\ )}$. However, not having to store the leading 1 ~~give~~ means the last digit represents multiples of 2^{-n}, not $2^{-(n-1)}$.

Example. In 52-digit binary floating point, for all in-range numbers x,

$$\frac{|x - fl(x)|}{|x|} \leq 2^{-53} \approx 10^{-15.9546}$$

In 23-digit binary floating point,

$$\frac{|x - fl(x)|}{|x|} \leq 2^{-24} \approx 10^{-7.22472}$$

Modern computers use IEEE-754 floating point to represent numbers, almost always as "binary64" or "double". On an x86 processor, these are stored as

$\underbrace{s}_{\text{sign bit}\ 1}$ $\underbrace{e_0 \quad\quad\quad e_{10}}_{\text{exponent bits}\ 11}$ $\underbrace{d_{52} \quad\quad\quad\quad\quad\quad d_1}_{\text{fraction bits}\ 52}$

Note that the digits in the fraction are stored in reverse ("little-endian") order!

The exponent is computed from $e_0, \ldots, e_{10}$ by writing the binary integer

$$(e_{10} \cdots e_0)_2 = \sum_{i=0}^{10} e_i \times 2^i$$

and subtracting 1023.

We never allow ($e_0 \cdots e_{10}$ all 0) or ($e_0 \cdots e_{10}$ all 1) so the exponent is between

$$(00000000001)_2 - 1023 = -1022$$

and

$$(11111111110)_2 - 1023 = 1023$$

This means that the in-range numbers greater than 0 are between

$$1 \times 2^{-1022} \approx 10^{-307.65} \quad \text{and} \quad (1.1\ldots1)_2 \times 2^{1023} \approx 10^{308.255}$$

Special cases. 1.

All <u>exponent</u> bits e_i are 0.

All fraction bits d_i are 0.

$s = 0$.

+0 "plus zero"

$s = 1$

−0 "minus zero"

Some fraction bits d_i are not zero.

A "subnormal" number

$$(-1)^s \times 2^{-1022} \times 0.d_1 \ldots d_{52}$$

Special case 2.

All exponent bits e_i are 1.

All fraction bit~~e~~s d_i are zero.

$s=0$

+Inf ~~&~~ "plus infinity"

$s=1$

-Inf "minus infinity"

Fraction bits have ~~are~~ $\cancel{d_1 d_2}\, d_2 \cdots d_{51} = 0,\ d_{52} = 1$.

$d_1 = 1$

qNaN "quiet Not a Number"

$d_1 = 0$

sNaN "signalling Not a Number"

General case.

In general, the exponent e is computed from the exponent bits and the number is

$$(-1)^s \times 1.d_1 \dots d_{52} \times 2^e$$

When Inf and NaN come up is more or less intuitive.

$$1/{\pm 0} = \pm \mathrm{Inf}$$

$$-1/{\pm 0} = \mp \mathrm{Inf}$$

$$\pm 0/{\pm 0} = \mathrm{NaN}$$

$$\pm \mathrm{Inf}/{\pm \mathrm{Inf}} = \mathrm{NaN}$$

$$\mathrm{Sqrt}(-1.0) = \mathrm{NaN}$$

$$+\mathrm{Inf} + (-\mathrm{Inf}) = \mathrm{NaN}$$

Most importantly, if a is any number,

$$a \times NaN = NaN$$

$$\frac{a}{NaN} \text{ and } \frac{NaN}{a} = NaN$$

$$a \pm NaN = NaN$$

$$f(NaN) = NaN$$

if f is a standard math function.

This means that an output of "NaN" can and does appear after a long sequence of calculations, if any intermediate step went bad.

Final note: The "single precision" or "float" or "binary32" format has

$$\underbrace{s}_{s}\ \underbrace{e_0 \qquad\qquad e_7}_{8 \text{ exponent bits}}\ \underbrace{d_{23} \qquad\qquad d_1}_{23 \text{ fraction bits}} = 32 \text{ bits}$$

which usually represent numbers ~~as~~ by

$$e = \left(\sum_{i=0}^{7} e_i \times 2^i\right) - 127$$

and the number is

$$\cancel{s\times} (-1)^s \times 1.d_1 \ldots d_{23} \times 2^e$$