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

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

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

\[ \ldots d_2d_1d_0 \ldots = b \]

where each \( d_i \) is 1 or 0 and

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

Examples.

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

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

0.\bar{1} \quad = \quad \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots = 1.

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

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

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

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

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

$$+1.0\ldots0 \times 2^{-1022} \approx 10^{-307.0\ldots0653}$$
and the largest positive number is 
\[ +1.1 \ldots 1 \times 10^{1023} \approx 10^{308} \]

Example: If \( n = 2 \), then

Fact. In \( n \)-digit binary floating point

\[ \frac{|x - f_1(x)|}{|x|} \leq 2^{-n-1} \]

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 \( \frac{1}{2} \times 2^e \).’ However, not having to store the leading 1 means the last digit represents multiples of \( 2^n \), not \( 2^{n-1} \).
Example. In $5_{10}$-digit binary floating point, for all in-range numbers $x$,

\[
\frac{|x-f_1(x)|}{1x1} \leq 2^{-543} \approx 10^{-15.9546}
\]

In $2_{10}$-digit binary floating point,

\[
\frac{|x-f_1(x)|}{1x1} \leq 2^{-24} \approx 10^{-7.22472}
\]

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

\[
\begin{array}{cccc}
S & E_0 & E_{10} & d_{52} \\
1 & 11 & 11 & 52
\end{array}
\]

sign bit exponent bits fraction bits

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

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

and subtracting 1023.

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

$$(000000000001)_2 - 1023 = -1022$$

and

$$(111111111110)_2 - 1023 = 1023$$

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 case 1.

All exponent 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 bits $d_i$ are zero.

$s = 0$

$+\text{Inf} \; \& \; "plus \; infinity"$

$s = 1$

$-\text{Inf} \; \; "minus \; infinity"$

Fraction bits $d_1 \ldots d_{51} = 0, d_{52} = 1$.

$d_1 = 1$

$q\text{NaN} \; \; "quiet \; Not \; a \; Number"$

$d_1 = 0$

$s\text{NaN} \; \; "signalling \; Not \; a \; Number"$
General case.

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

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

When $\text{Inf}$ and $\text{NaN}$ come up is more or less intuitive.

$$\frac{1}{\pm 0} = \pm \text{Inf}$$

$$-\frac{1}{\pm 0} = \mp \text{Inf}$$

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

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

$$\sqrt{(-1.0)} = \text{NaN}$$

$$+\text{Inf} + (-\text{Inf}) = \text{NaN}$$
Most importantly, if \( a \) is any number,

\[
\frac{a}{NaN} \quad \text{and} \quad \frac{NaN}{a} = NaN
\]

\[
a \pm NaN = NaN
\]

\[
\& f(NaN) = NaN
\]

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

\[ \frac{s}{5} \begin{array}{cccc}
  e_0 & e_7 & d_{23} & d_{4}\n\end{array} \]

5 8 exponent bits 23 fraction bits = 32 bits

which usually represent numbers \( e \) by

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

and the number is

\[ (-1)^s \times 1.d_1 d_2 \ldots d_{23} \times 2^e \]