Fixed-point and floating-point representations of numbers

A fixed-point representation of a number may be thought to consist of 3 parts: the sign field, integer field, and fractional field. One way to store a number using a 32-bit format is to reserve 1 bit for the sign, 15 bits for the integer part and 16 bits for the fractional part. A number whose representation exceeds 32 bits would have to be stored inexactly.

On a computer, 0 is used to represent + and 1 is used to represent −.

Example. The 32-bit string

$$1 \mid 000000000101011 \mid 1010000000000000$$

represents $(-101011.101)_2 = -43.625$.

The fixed point notation, although not without virtues, is usually inadequate for numerical analysis as it does not allow enough numbers and accuracy.

Example. In the format just discussed, the largest number is

$$0 \mid 111111111111111 \mid 1111111111111111$$

or $(2^{15} - 1) + (1 - 2^{-16}) = 2^{15}(1 - 2^{-31}) \approx 32768$, and the smallest positive number is

$$0 \mid 000000000000000 \mid 0000000000000001$$

or $2^{-16} \approx 0.000015$. Note that 2^{-16} is precisely the gap between two adjacent fixed-point numbers.

The floating-point notation is by far more flexible. Any $x \neq 0$ may be written in the form

$$\pm(1.b_1b_2b_3...)_2 \times 2^n,$$

called the *normalized representation* of x. The normalized representation is achieved by choosing the exponent n so that the binary point "floats" to the position after the first nonzero digit. This is the binary version of scientific notation.

To store a normalized number in 32-bit format one reserves 1 bit for the sign, 8 bits for the signed exponent, and 23 bits for the portion $b_1b_2b_3...b_{23}$ of the fractional part of the number. The leading bit 1 is not stored (as it is always 1 for a normalized number) and is referred to as a "hidden bit".

The 8-bit exponent field is used to store integer exponents $-126 \leq n \leq 127$.
We will discuss later how exactly this is done.

Example. The 32-bit string

$$1 \mid 8 \text{ bits storing n=5} \mid 10101100000000000000000$$

represents $(-1.101011)_2 \times 2^5 = (-110101.1)_2 = -53.5$.

Example. The 32-bit word

0 | 8 bits storing n=0 | 00000000000000000000000

represents $(1.0)_2 = 1$.

Example. The largest normalized number that fits into 32 bits is

0 | 8 bits storing n=127 | 11111111111111111111111

or $(1.11111111111111111111111)_2 \times 2^{127} = (2^{24} - 1)2^{104} \approx 3.40 \times 10^{38}$.
The smallest normalized positive number that fits into 32 bits is

0 | 8 bits storing n=-126 | 00000000000000000000000

or $(1.00000000000000000000000)_2 \times 2^{-126} = 2^{-126} \approx 1.18 \times 10^{-38}$.

The *precision* of a floating-point format is the number of positions reserved for binary digits plus one (for the hidden bit). In the examples considered here the precision is 23+1=24.

One says that x is a floating-point number if it can be represented exactly using a given floating-point format. For instance, $1/3 = (0.010101\ldots)_2$ cannot be a floating-point number as its binary representation is nonterminating.

The gap between 1 and the next normalized floating-point number is known as *machine epsilon*. In our setting, this gap is $(1 + 2^{-23}) - 1 = 2^{-23}$. Note that this is not the same as the smallest positive floating-point number. Unlike in the fixed-point scenario, the spacing between the floating-point numbers is not uniform, but varies from one dyadic interval $[2^n, 2^{n+1})$ to another. As we move away from the origin, the spacing becomes less dense: