

Computer Representation of Floating-Point Numbers

A classic computer nerd T-shirt reads:

“There are 10 types of people in the world.
Those who understand binary and those who don’t.”

We need to be among those who do understand, because the use of a binary representation of numbers has important implications for computational programming. If we use N bits (a bit is either 0 or 1) to store an integer, we can only represent 2^N different integers. Since the sign takes up the first bit (in general), we have $N - 1$ bits for the absolute value, which is then in the range $[0, 2^{N-1} - 1]$. A C++ `int` uses 32 bits = 4 bytes, which means the maximum integer is $2^{31} \approx 2 \times 10^9$ (an `unsigned int` goes up to $2^{32} \approx 4 \times 10^9$). This doesn’t seem very large when you consider that the ratio of the size of the universe to the size of a proton is about 10^{24} [1]! [Note: Comair had computer problems in Christmas, 2004 caused by a computer using 16 bit integers, which limited the number of scheduling changes per month to 2^{15} . That was exceeded because of storms and the whole software system crashed!]

A unique, well-defined, but in general approximate representation is used for “floating point” numbers (as opposed to the representation for integers). It is a form of “normalized scientific notation”, where in the decimal version 35.216 is represented as 0.35216×10^2 or, more generally,

$$x = \pm r \times 10^n \quad \text{with} \quad 1/10 \leq r < 1, \quad (1)$$

with r called the “mantissa” and n the “exponent” (note that the first digit in r must be nonzero except when $x = 0$). Here is the basic form for the binary equivalent:

$$(\text{any number}) = (-1)^{\text{sign}} \times (\text{base 2 mantissa}) \times 2^{[(\text{exponent field}) - \text{bias}]}, \quad (2)$$

and the computer stores the sign, base 2 mantissa, and the exponent field. The *bias* serves to keep the stored exponent positive, so that we don’t need to store its sign, saving a bit.

Let’s look at an analogous base 10 representation with six digits kept in the mantissa, one digit in the exponent, and a bias of 5:

$$-\frac{4}{3} = -1.33\bar{3} \doteq (-1)^1 \times (.133333) \times 10^{[6-5]}, \quad (3)$$

where we would store the 1 (for the sign), 133333, and 6. What are the largest and smallest possible numbers, and what is the precision? The exponent can be as large as 9 or as small as 0 so the numbers range from 0.1×10^{-5} to $.999999 \times 10^4$. The precision is six decimal digits. This means that while we can represent $x = 3500 = 0.35 \times 10^{[9-5]}$ and $y = 0.0021 = 0.21 \times 10^{[2-5]}$, if we try to add them and store the result, we find that $x + y = x$!

In base 2, the mantissa for a single-precision float takes the form

$$\text{mantissa} = m_1 \times 2^{-1} + m_2 \times 2^{-2} + \dots + m_{23} \times 2^{-23}, \quad (4)$$

where each m_i is either 0 or 1, so there are 23 bits to store (each of the m_i 's is either 0 or 1). For the sign we need 1 bit. If we use 8 bits for the exponent, that is a total of 32 bits or 4 bytes (i.e., 1 byte = 8 bits). To have a unique representation with all numbers having roughly the same precision, we require $m_1 = 1$ (except for 0) since, for example, we could otherwise represent $1/2$ as both $(1 \times 2^{-1}) \times 2^0$ and $(1 \times 2^{-2}) \times 2^1$. (Thus, m_1 doesn't have to be stored in practice and we could, in principle, pick up an extra bit of storage.) The largest number stored would be

$$\underbrace{0}_{\text{sign}} \quad \underbrace{1111 \ 1111}_{\text{exponent}} \quad \underbrace{1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111}_{\text{mantissa}} \quad (5)$$

and the smallest number would be

$$\underbrace{0}_{\text{sign}} \quad \underbrace{0000 \ 0000}_{\text{exponent}} \quad \underbrace{1000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000}_{\text{mantissa}} \quad (6)$$

To figure out the actual range of numbers that can be stored, we also need to specify the bias, which is $127_{10} = 0111 \ 1111_2$ for single precision. This means that the number 0.5 is stored as [1]

$$\underbrace{0}_{\text{sign}} \quad \underbrace{0111 \ 1111}_{\text{exponent}} \quad \underbrace{1000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000}_{\text{mantissa}} \quad (7)$$

It also implies that (you verify these!)

$$\text{largest number:} \quad 2^{128} \approx 3.4 \times 10^{38} \quad (8)$$

$$\text{smallest number:} \quad 2^{-128} \approx 2.9 \times 10^{-39} \quad (9)$$

$$\text{precision:} \quad 6\text{--}7 \text{ decimal places (1 part in } 2^{23}\text{)} \quad (10)$$

If a single-precision number becomes larger than the largest number, we have an *overflow*. If it becomes smaller than the smallest number, we have an *underflow*. An overflow is typically a disaster for our calculation while an underflow is usually just set to zero automatically without a problem. For a double precision number, eight bytes or 64 bits are used, with 1 for the sign, 52 for the mantissa, 11 for the exponent, and a bias of 1023. *Exercise: Figure out the expected range of numbers and the precision for doubles.*

[Note: This discussion of floating point numbers is based closely on Refs. [1] and [2]. The actual implementation for the computers we use is the IEEE standard for floating-point numbers, which introduces some slight differences.]

Most floating-point numbers cannot be represented exactly (those that can are called “machine numbers”). For example, the decimal 0.25 is a machine number but 0.2 is not! We can use Mathematica to find the first digits of the base 2 representation of 0.2:

```
BaseForm[0.2, 2]
```

yields $0.0011001100110011001101_2$ and the pattern actually repeats indefinitely (can you do base 2 long division to derive this by hand?). Now suppose we only had enough storage to keep 0.00110011.

As a decimal, this is 0.19921875. So the actual number deviates from the computer representation. The maximum deviation is related to the *machine precision*.

Any number z is related to its machine number computer representation z_c by

$$z_c = z(1 + \epsilon) \quad \text{with} \quad |\epsilon| \leq \epsilon_m, \quad (11)$$

where ϵ_m is the machine precision, which is defined as the largest number ϵ for which $1 + \epsilon = 1$ in a given representation (e.g., float or double). *Note that the machine precision ϵ_m is not the smallest floating-point number that can be represented.* The former depends on the number of bits in the mantissa while the latter depends on the number of bits in the exponent [3].

In MATLAB, numbers are stored in double precision, which means they will have about 16 decimal places of accuracy. Repeated operations (e.g., multiplications or subtractions) can *accumulate* errors, depending on how numbers are combined. We see this when taking numerical derivatives, integrals, and so on.

References

- [1] R.H. Landau and M.J. Paez, *Computational Physics: Problem Solving with Computers* (Wiley-Interscience, 1997).
- [2] M. Hjorth-Jensen, *Computational Physics*. These are notes from a course offered at the University of Oslo. See the 780.20 webpage for links.
- [3] W. Press *et al.*, *Numerical Recipes in C* (Cambridge, 1992). Individual chapters are available online from <http://lib-www.lanl.gov/numerical/bookcpdf.html>. There are also versions for Fortran and C++.