Lecture `float`: Machine numbers

August 29, 2022

Summary: These notes describe how floating point numbers are represented on a digital computer, with an example of a normalized floating point number worked out in detail.

References: Part of what follows stems from a treatment given by Prof. Monika Nitsche. It also follows and borrows examples from the first edition of T. Sauer's textbook *Numerical Analysis*, Section 0.3.

Conversion between decimal and binary

We review the process of converting a base-10 representation of a real number $x \in \mathbb{R}$ to its base-2 representation, and vice versa. These processes are needed to understand the machine representation.

base-10 and base-2 representations

As you know, the base-10 or decimal representation is

$$x = (d_N \cdots d_2 d_1 d_0 . d_{-1} d_{-2} \cdots)_{10},$$

where each *digit* d_k is an integer obeying $0 \le d_k \le 9$, and the representation corresponds to the number

$$x = d_N 10^N + \cdots + d_2 10^2 + d_1 10^1 + d_0 10^0 + d_{-1} 10^{-1} + d_{-2} 10^{-2} + \cdots .$$

For example, consider

$$\begin{aligned} &x = (931.84)_{10} \text{ with } N = 2, \quad d_2 = 9, d_1 = 3, d_0 = 1, \quad d_{-1} = 8, d_{-2} = 4, \\ &x = (8782.\overline{3})_{10} \text{ with } N = 3, \quad d_3 = 8, d_2 = 7, d_1 = 8, d_0 = 2, \quad 3 = d_{-1} = d_{-2} = \cdots , \end{aligned}$$

which we would customarily write simply as $x = 931.84$ and $x = 8782.\overline{3}$. The base-2 or binary representation

$$x = (b_M \cdots b_2 b_1 b_0 . b_{-1} b_{-2} \cdots)_2,$$

where each binary digit or *bit* b_k obeys $0 \le b_k \le 1$, similiarly represents the number

$$x = b_M 2^M + \cdots + b_2 2^2 + b_1 2^1 + b_0 2^0 + b_{-1} 2^{-1} + b_{-2} 2^{-2} + \cdots .$$

For example, consider

$$x = (101.11)_2 \text{ with } M = 2, \quad b_2 = 1, b_1 = 0, b_0 = 1, \quad b_{-1} = 1, b_{-2} = 1.$$

If we are just dealing with decimal and binary representations (and not also, say, base-8), then there's no ambiguity in writing $x = 532.34567$; it has to be base-10. However, is $x = 101.11$ a binary or decimal representation? Here we should be careful and either write $x = (101.11)_{10}$ or $x = (101.11)_2$.

Conversion

Each representation of a number has an *integer part* and a *fractional part*. For example, the integer and fractional parts of $x = 532.34567$ are $x_{\text{int}} = 532$ and $x_{\text{frac}} = 0.34567$, respectively. Similarly, the integer and fractional parts of $x = (101.11)_{10}$ are $x_{\text{int}} = (101)_2$ and $x_{\text{frac}} = (0.11)_2$, respectively. For both decimal-to-binary (base-10 to base-2) and binary-to-decimal (base-2 to base-10) conversions, integer parts go to integer parts and fractional parts go to fractional parts. We consider these separately.

Integer part

The integer part of $x = 532.34567$ is $x_{\text{int}} = 532$. We convert 532 to its binary representation via the following recipe.

```
532/2 = 266 remainder 0      get b0
266/2 = 133 remainder 0      get b1
133/2 =  66 remainder 1      get b2
 66/2 =  33 remainder 0      get b3
 33/2 =  16 remainder 1      get b4
 16/2 =   8 remainder 0      get b5
  8/2 =   4 remainder 0      get b6
  4/2 =   2 remainder 0      get b7
  2/2 =   1 remainder 0      get b8
  1/2 =   0 remainder 1      get b9   When 0 appears in division, bM has been found.
```

Thus $532 = (1000010100)_2$. Going the other way is easier. For example, as expected

$$(1000010100)_2 = 1\cdot 2^9 + 0\cdot 2^8 + 0\cdot 2^7 + 0\cdot 2^6 + 0\cdot 2^5 + 1\cdot 2^4 + 0\cdot 2^3 + 1\cdot 2^2 + 0\cdot 2^1 + 0\cdot 2^0 = 512 + 16 + 4 = 532.$$

Fractional part

Conversion of fractional parts is typically more difficult. Let's first consider decimal-to-binary examples. The fractional part of $x = 9.4$ is $x_{\text{frac}} = 0.4$. Let's convert 0.4 to binary. Here is the recipe.

```
0.4 * 2 = 0.8  plus 0        get b_(-1)
0.8 * 2 = 0.6  plus 1        get b_(-2)
0.6 * 2 = 0.2  plus 1        get b_(-3)
0.2 * 2 = 0.4  plus 0        get b_(-4)
0.4 ........................ pattern will now infinitely repeat
```

So $0.4 = (0.\overline{0110})_2$. Since the integer part of 9.4 is clearly $9 = (1001)_2$, in total $9.4 = (1001.\overline{0110})_2$. Let's consider another example. The whole fraction $x = 44/7 = 6\frac{2}{7}$ has integer part $x_{\text{int}} = 6 = (110.0)_2$ and fractional part $x_{\text{frac}} = 2/7$. When converting this fractional part to binary, work with fractions as follows (i.e. don't first convert 2/7 to a decimal expansion).

```
2/7 * 2 = 4/7  plus 0        get b_(-1)
4/7 * 2 = 1/7  plus 1        get b_(-2)
1/7 * 2 = 2/7  plus 0        get b_(-3)
2/7 ........................ pattern will now infinitely repeat
```

So $2/7 = (0.\overline{010})_2$, and in total $44/7 = (110.\overline{010})_2$. Conversion of a fractional part given just as an infinitely repeating decimal expansion may be difficult, but, in principle, is straightforward if it's possible to write the expansion as a fraction. For example, $0.\overline{09} = \frac{1}{11}$, and

```
 1/11 * 2 =  2/11  plus 0    get b_(-1)
 2/11 * 2 =  4/11  plus 0    get b_(-2)
 4/11 * 2 =  8/11  plus 0    get b_(-3)
```

```
 8/11 * 2 =  5/11  plus 1    get b_(-4)
 5/11 * 2 = 10/11  plus 0    get b_(-5)
10/11 * 2 =  9/11  plus 1    get b_(-6)
 9/11 * 2 =  7/11  plus 1    get b_(-7)
 7/11 * 2 =  3/11  plus 1    get b_(-8)
 3/11 * 2 =  6/11  plus 0    get b_(-9)
 6/11 * 2 =  1/11  plus 1    get b_(-10)
 1/11 ..................... pattern will now infinitely repeat
```

So we find $0.\overline{09} = (0.\overline{0001011101})_2$.

Binary-to-decimal conversion of a fractional part is again easy if the binary expansion is finite. e.g.

$$(0.1011)_2 = 1 \cdot \tfrac{1}{2} + 0 \cdot \tfrac{1}{4} + 1 \cdot \tfrac{1}{8} + 1 \cdot \tfrac{1}{16} = \tfrac{11}{16}.$$

If the binary expansion doesn't truncate, then you have an infinite series to sum, and that of course may be difficult. You might be able to exploit the explicit formula for a geometric series. For example,

$$(0.\overline{1})_2 = \sum_{k=1}^{\infty} (\tfrac{1}{2})^k = -1 + \sum_{k=0}^{\infty} (\tfrac{1}{2})^k = -1 + \frac{1}{1-\frac{1}{2}} = 1.$$

Sometimes other tricks are possible. For example, $x = (111.\overline{110})_2$ has integer part $x_{\text{int}} = (111.0)_2 = 7$ and fractional part $x_{\text{frac}} = (0.\overline{110})_2$. To convert the fractional part to base-10, consider

$$8x_{\text{frac}} = (110.\overline{110})_2 \implies 7x_{\text{frac}} = 8x_{\text{frac}} - x_{\text{frac}} = (110.0)_2 = 6 \implies x_{\text{frac}} = 6/7.$$

Therefore, in total $(111.\overline{110})_2 = 7\frac{6}{7} = 55/7$.

Floating point storage and machine precision

There are two common representations for machine numbers, single precision with 4 bytes (32 bits) for each number and double precision with 8 bytes (64 bits) for each number. In scientific computations, single precision is typically only used in specialized applications where there is no danger of a loss of precision. So here we will only consider double precision numbers. Double precision is the default used by MATLAB, the type `double` in C and C++, and the type `real*8` or `double precision` in Fortran. The representation used almost universally today is an IEEE standard. IEEE stands for *Institute of Electrical and Electronics Engineers* (see `www.ieee.org`) and IEEE 754, described here, is the standard for floating point numbers.

Floating point numbers are represented in base 2, using a *mantissa*,[1] an exponent, and a sign bit. In IEEE double precision, 64 bits are stored to represent a number: s (sign bit), $e_{10}, \ldots, e_1, e_0$ (exponent), and $b_1, b_2, \ldots, b_{52}$ (mantissa), where each of the bits has value 0 or 1, as shown in Fig. 1. A number F is formed using the e_i in base 2:

$$F = (e_{10}, \ldots, e_1, e_0)_2 = e_{10}2^{10} + \cdots + e_1 2^1 + e_0 2^0,$$

which corresponds to the values

$$0 \le F \le 2047.$$

To see that F can attain 2047, notice that $(11111111111)_2 + (1)_2 = (100000000000)_2 = 2^{11} = 2048$. Therefore, $(11111111111)_2 = 2048 - 1 = 2047$. The value that the 64 bits represent depends on the value of F and the mantissa as follows.

[1] Latin for *an addition*.

Normalized numbers

If $1 \le F \le 2046$, then the 64 bits represent the real number

$$V = (-1)^s \, (1.b_1 b_2 b_3 \ldots b_{52})_2 \times 2^{F-1023}. \tag{1}$$

The sign bit s determines whether the number is positive or negative. In base 10 the normalized leading factor is[2]

$$(1.b_1 b_2 \ldots b_{52})_2 = 1 + b_1 2^{-1} + b_2 2^{-2} + b_3 2^{-3} + \ldots b_{52} 2^{-52}.$$

Therefore, the largest normalized number is

$$\begin{aligned} V_{\max} = V_{\max,\text{norm}} &= (1.1111 \ldots 1)_2 \times 2^{1023} = \left(\sum_{n=0}^{52} 2^{-n} \right) 2^{1023} = \frac{1 - 2^{-53}}{1 - \frac{1}{2}} 2^{1023} \\ &= 2^{971}(2^{53} - 1) \simeq 1.7977 \times 10^{308}. \end{aligned}$$

Notice that the largest *normalized* floating point number is also the largest floating point number. The smallest normalized number is

$$V_{\min,\text{norm}} = 1 \times 2^{-1022} \simeq 2.2251 \times 10^{-308}.$$

Unnormalized (or subnormal) numbers

If $F = 0$, then

$$V = (-1)^s \, (0.b_1 b_3 b_4 \ldots b_{52})_2 \times 2^{-1022}$$

The motivation for switching the representation when $F = 0$ is to squeeze in more numbers near zero; see Fig. 2. What are largest and smallest nonzero unnormalized numbers?

$$V_{\max,\text{unnorm}} = (0.111 \ldots 1)_2 \times 2^{-1022}.$$

This is the next number with 52 binary digits smaller than $V_{\min,\text{norm}}$. The smallest nonzero unnormalized number is

$$V_{\min} = V_{\min,\text{unnorm}} = (0.\underbrace{00 \cdots 01}_{52 \text{ bits}})_2 \times 10^{-1022} = 2^{-52} \times 2^{-1022} \simeq 4.9407 \times 10^{-324}.$$

Notice that the smallest nonzero *unnormalized* floating point number is also the smallest nonzero floating point number. Finally, the number 0 itself is representable as an unnormalized floating point number, namely as $(0.0 \cdots 0)_2 \times 2^{-1022}$.

Not-a-Number (NaN) and Infinity

If $F = 2047$ and the mantissa is non-zero, then V has the value NaN. This is produced by operations whose result is not defined or cannot be represented by real numbers, such as division by zero and the square root of a negative number (but MATLAB handles complex numbers just fine). If $F = 2047$ and the mantissa is 0, then $V = (-1)^s \infty$.

[2] Before we used the notation

$$x = (b_M \cdot b_2 b_1 b_0 . b_{-1} b_{-2} \cdots)_2,$$

with negative indices on the b's to the right of the binary point. Now, we are changing notation and have positive indices on the b's to the right of the binary point. Since we are always *normalizing* the number; that is, using scientific notation to start with $(1_\bullet \cdots)_2 \times 2^{F-1023}$, we won't need bits to the left of the binary point.

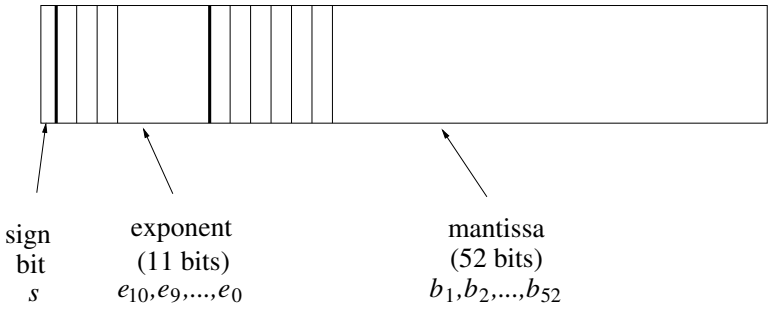

Figure 1: Format of the IEEE standard for double precision floating point numbers. Eight bytes (64 bits) of storage is required.

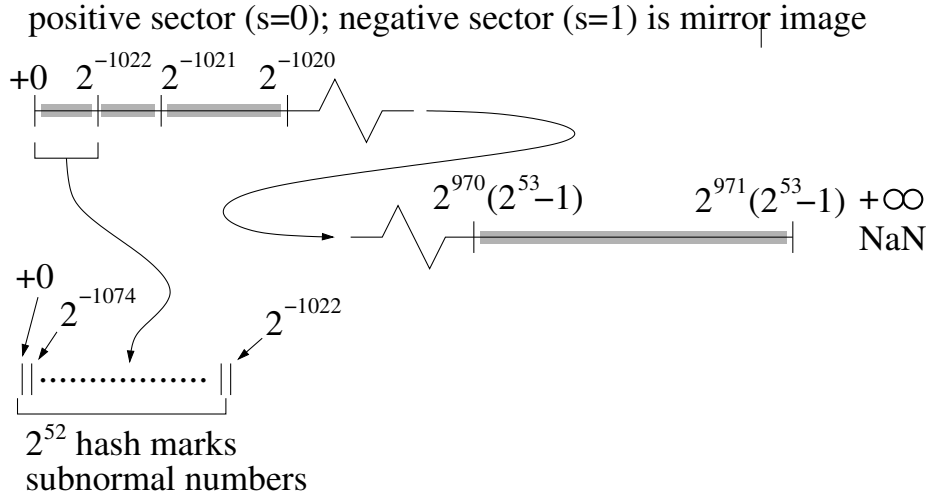

Figure 2: Double precision floating point number system.

Overflow, Underflow, and Machine precision

Let us perform a few experiments in MATLAB to confirm our results.

From above, we know that the largest representable double precision number is $V_{\max} = 2^{971}(2^{53}-1)$. What happens if we try to create a larger number?

```
>> format compact
>> x = 2^(971)*(2^(53)-1)
x =
  1.7977e+308
>> 2*x
ans =
   Inf
```

This is an example of overflow. If a valid operation leads to a value too large to be represented, then the result is set to ∞, with the appropriate sign, if possible. An overflow exception is usually raised by the processor. This exception usually results in the program being aborted. If the program proceeds, then all future calculations with ∞ will return NaN.

From above, we know that the smallest representable double precision number is $V_{\min} = 2^{-1074}$. In MATLAB, we try to compute a smaller number.

```
>> x = 2^(-1074)
x =
  4.9407e-324
>> x/2
ans =
     0
```

This is an example of underflow. If a valid operation leads to a value too small to be represented, then the result is set to zero. An underflow exception is raised by the processor. Underflow is usually not that serious —it means a very small number has been replaced by zero, and thus underflow exceptions are usually ignored by the program.

Another important number is the *machine precision* or *machine epsilon*, denoted by ε or sometimes $\varepsilon_{\text{mach}}$. Machine precision is related to the distance between 1 and the next largest floating point number that can be represented in double precision. The exact definition of $\varepsilon_{\text{mach}}$ is the smallest positive number such that

$$1 + \varepsilon_{\text{mach}} \neq 1,$$

where the addition here is the computer implementation (a floating point addition). Any number smaller than $\varepsilon_{\text{mach}}$, when added to 1, yields a result which, upon rounding to the closest 64-bit floating point number, yields 1 back again. In double precision[3] $\varepsilon_{\text{mach}} = 2^{-52} \simeq 2.2204 \times 10^{-16}$. In Matlab, machine precision is given by `eps`, and the largest and smallest *normalized* floating point numbers are `realmin` and `realmax`.

Example from Sauer's textbook

As an illustrative example, we consider the IEEE floating point representation of the number 9.4, assuming double precision as is the case in MATLAB. Note that 9.4 is not anywhere close to being a subnormal number, and so it falls within the range represented by *normalized* floating point numbers. Indeed, clearly $x = 9.4$ obeys $2^{-1022} \leq |x| \leq (2 - 2^{-52})2^{1023}$.

Binary expression. We saw above that

$$(9.4)_{10} = (1001.\overline{0110})_2. \tag{2}$$

Via expansion and a shift of the binary point, we then write (here, and in all subsequent formula, numbers which involve boxes are expressed in base-2)

$$(1001.\overline{0110})_2 = \\ +1.\underbrace{\boxed{001\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110}}_{52\text{ bits}}110\ \ldots \times 2^3. \tag{3}$$

This is, of course, an exact representation of 9.4, just the one in base-2. However, for storage in computer memory this infinite expansion must be truncated and (possibly) modified. This process gives rise to the floating point number fl(9.4) corresponding to 9.4.

Corresponding floating point number. The 53rd bit, the one which lies just outside the box above, proves most crucial in obtaining fl(9.4), but subsequent bits (the 54th, etc.) also play a role. Here is the general rule, assuming $2^{-1022} \leq |x| \leq (2 - 2^{-52})2^{1023}$ (for a subnormal number, $0 \leq |x| \leq$

[3]You may also encounter the definition $\varepsilon_{\text{mach}} = 2^{-53} \simeq 1.1102 \times 10^{-16}$, a variant based on the bound for relative rounding errors. We will use Sauer's $\varepsilon_{\text{mach}} = 2^{-52}$.

$(1-2^{-52})2^{-1022}$, the binary expansion is expressed a little differently.) To convert a number x into its IEEE floating point representation, we first express the number in a normalized binary expansion as above. Next, we have to extract from this expansion the appropriate floating point number. This extraction is achieved as follows.

> **IEEE rounding to the nearest rule for double precision.** (i) If the 53rd bit to the right of the binary point is 0, then round down, i. e. truncate the 53rd and all remaining bits. (ii) If the 53rd bit to the right of the binary point is 1, then consider the following two cases: (a) if at least one bit after and including the 54th is 1, then round up, i. e. add 1 to the 52nd bit; (b) if the 54th and all trailing bits are 0, then add 1 to the 52nd bit if and only if the 52nd bit is 1.

Possibilities (i) and (iia) correspond to rounding the number x in question to the *nearest* floating point number, which is then called $\mathrm{fl}(x)$. For (iib), the number x lies exactly halfway between adjacent floating point numbers. In this case, we round up or down based on which gives a 0 for the 52nd bit. The idea here is to avoid a statistical bias in the rounding; whether to round or not in the exceptional case (iib) depends on the number encountered.

Our example falls into case (iia) of the rule; we therefore round up to get

$$\mathrm{fl}(9.4) = +1.\underbrace{\boxed{001\ \ 0110\ \ 0110\ \ 0110\ \ 0110\ \ 0110\ \ 0110\ \ 0110\ \ 0110\ \ 0110\ \ 0110\ \ 0110\ \ 0110\ \ 1}}_{52\text{ bits}}\times 2^3. \tag{4}$$

Which base-10 number do we have now? In modifying 9.4 to get $\mathrm{fl}(9.4)$, we took two steps. First, from 9.4 we subtracted (this is the truncation)

$$(1.10\overline{0110})_2 \times 2^{-53} \times 2^3 = (0.\overline{0110})_2 \times 2^{-48} = (0.4)_{10} \times 2^{-48},$$

with the last equality stemming from our calculations above. Second, to 9.4 we added (this is the rounding up)

$$2^{-52} \times 2^3 = 2^{-49}.$$

Therefore, in base-10 representation

$$\mathrm{fl}(9.4) = 9.4 - 0.4 \times 2^{-48} + 2^{-49} = 9.4 + 0.2 \times 2^{-49}. \tag{5}$$

For this example, notice that

$$\frac{|\mathrm{fl}(9.4) - 9.4|}{9.4} = \frac{0.2 \times 2^{-49}}{9.4} = \frac{1}{47} \times 2^{-49} = \frac{8}{47} \times 2^{-52} < \frac{1}{2}\varepsilon_{\mathrm{mach}}, \tag{6}$$

where we have used Sauer's convention for $\varepsilon_{\mathrm{mach}} = 2^{-52}$. (However, be aware that some authors define $\varepsilon_{\mathrm{mach}} = 2^{-53}$.) This example illustrates a general property of the IEEE format [cf. Eq. (0.9) from Sauer]:

$$\frac{|\mathrm{fl}(x) - x|}{|x|} \leq \frac{1}{2}\varepsilon_{\mathrm{mach}}. \tag{7}$$

Machine storage. Finally, let us consider how $\mathrm{fl}(9.4)$ would be stored in memory. From the above, we find the following result.

mantissa for fl(9.4): $\boxed{001\ \ 0110\ \ 0110\ \ 0110\ \ 0110\ \ 0110\ \ 0110\ \ 0110\ \ 0110\ \ 0110\ \ 0110\ \ 0110\ \ 0110\ \ 1}$

The true exponent for $\mathrm{fl}(9.4)$ is $3 = F - 1023$, so that

$$\begin{aligned} F &= (1026)_{10} \\ &= (1024)_{10} + (2)_{10} \\ &= (10000000000)_2 + (10)_2 \\ &= (10000000010)_2. \end{aligned}$$

Therefore, the exponent is stored as follows.

$$\text{exponent for fl(9.4): } \boxed{10000000010}$$

The sign bit is 0 (which corresponds to +), and in all we find that fl(9.4) will be stored as follows (sign+exponent+mantissa).

$$\begin{aligned} &\boxed{0}\,\boxed{10000000010}\,\boxed{001\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 1} \\ =&\boxed{0100\ 0000\ 0010\ 0010\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1101} \end{aligned} \tag{8}$$

This is a somewhat unwieldy expression, and —to the author's knowledge— MATLAB does not allow one to directly see the 64-bit (8 byte) machine representation of a floating point number. However, MATLAB does have a `format hex` option, which is nearly as good for getting the machine representation.

Hexidecimal representation. Here is a key for elementary translation between base-16 (hexidecimal) and base-2 numbers.

$$\begin{array}{llll} (0)_{16} = (0000)_2 & (4)_{16} = (0100)_2 & (8)_{16} = (1000)_2 & (\text{c})_{16} = (1100)_2 \\ (1)_{16} = (0001)_2 & (5)_{16} = (0101)_2 & (9)_{16} = (1001)_2 & (\text{d})_{16} = (1101)_2 \\ (2)_{16} = (0010)_2 & (6)_{16} = (0110)_2 & (\text{a})_{16} = (1010)_2 & (\text{e})_{16} = (1110)_2 \\ (3)_{16} = (0011)_2 & (7)_{16} = (0111)_2 & (\text{b})_{16} = (1011)_2 & (\text{f})_{16} = (1111)_2 \end{array}$$

In Eq. (8) we have already split up the 64-bit machine representation of fl(9.4) in 4-bit chunks. Contiguously translating each chunk into "hex", we see that fl(9.4) corresponds to 4022cccccccccccd. *This is not the hexidecimal representation for* 9.4 *or* fl(9.4), *rather it is a mnemonic for recalling the machine storage.* We can check our hex format in MATLAB:

```
>> format hex
>> 9.4

ans =

   4022cccccccccccd
```

Floating point arithmetic

Consider the basic arithmetic operations $+, -, \cdot, /$. Each of these has a corresponding implementation in IEEE arithmetic, denoted here by $\oplus, \ominus, \odot, \oslash$. Although, we need not fully understand the details, the key point is that *the computer makes mistakes!* For example, even if $\text{fl}(x) = x$ and $\text{fl}(y) = y$ (both x and y are already floating point numbers), it need not be true that the machine addition $x \oplus y = x + y$; indeed, the true sum $x + y$ may not even be a floating point number. That is, there's no guarantee that $\text{fl}(x + y) = x + y$. The following is a model for what we can expect:

$$x \oplus y = (x + y)(1 + \mu),$$

where $|\mu| \le \varepsilon_{\text{mach}}$. Therefore, provided $x + y \neq 0$, we have

$$\left| \frac{(x \oplus y) - (x + y)}{x + y} \right| = |\mu| \le \varepsilon_{\text{mach}}.$$

So the addition is good in a relative sense. Likewise we may expect

$$x \ominus y = (x - y)(1 + \lambda), \qquad x \otimes y = (x \times y)(1 + \delta), \qquad x \oslash y = (x/y)(1 + \alpha),$$

where $|\lambda|$, $|\delta|$, and $|\alpha|$ are all $\le \varepsilon_{\text{mach}}$.