Chapter 1

Floating point arithmetic

Introduction

When we study numerical algorithms in the next chapters, we assume implicitly that the operations involved are performed exactly. On a computer, however, only a subset of the real numbers can be stored and, consequently, many arithmetic operations are performed only approximately. This is the source of the so-called *round-off errors*. The rest of this chapter is organized as follows.

- In Section 1.1, we discuss the binary representation of real numbers.
- In Section 1.2, we describe the set of floating point numbers that can be represented in the usual floating point formats;
- In Section 1.3 we explain how arithmetic operations between floating point numbers behave. We insist in particular on the fact that, in a calculation involving several successive arithmetic operations, the result of each intermediate operation is stored as a floating point number, with a possible error.
- In Section 1.4, we briefly present how floating point numbers are encoded according to the IEEE 754 standard, widely accepted today. We discuss also the encoding of special values such as `Inf`, `-Inf` and `NaN`.
- Finally, in Section 1.5, we present the standard integer formats and their encoding.

In order to completely describe floating-point arithmetic, one would in principle need to also discuss the conversion mechanisms between different number formats, as well as a number of edge cases. Needless to say, a comprehensive discussion of the subject is beyond the scope of this course; our aim in this chapter is only to introduce the key concepts.

1.1 Binary representation of real numbers

Given any integer number $\beta > 0$, called the *base*, a real number x can always be expressed as a finite or infinite series of the form

$$x = \pm \sum_{k=-n}^{\infty} a_k \beta^{-k}, \qquad a_k \in \{0, \dots, \beta - 1\}. \tag{1.1}$$

The number x may then be denoted as $\pm(a_{-n}a_{-n+1}\dots a_{-1}a_0.a_1a_2\dots)_\beta$, where the subscript β indicates the base. This numeral system is called the *positional notation* and is universally used today, both by humans (usually with $\beta = 10$) and machines (usually with $\beta = 2$). If the base β is omitted, it is always assumed in this course that $\beta = 10$ unless otherwise specified – this is the *decimal* representation. The *digits* $a_{-n}, a_{-n+1}, \dots$ are also called *bits* if $\beta = 2$. In computer science, several bases other than 10 are regularly employed, for example the following:

- Base 2 (binary) is the usual choice for storing numbers on a machine. The binary format is convenient because the digits have only two possible values, 0 or 1, and so they can be stored using simple electrical circuits with two states. We employ the binary notation extensively in the rest of this chapter. Notice that, just like multiplying and dividing by 10 is easy in base 10, multiplying and dividing by 2 is very simple in base 2: these operations amount to shifting all the bits by one position to the left or right, respectively.

- Base 16 (hexadecimal) is sometimes convenient to represent numbers in a compact manner. In order to represent the values 0-15 by a single digit, 16 different symbols are required, which are conventionally denoted by $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$. With this notation, we have $(FF)_{16} = (255)_{10}$, for example.

  The hexadecimal notation is often used in programming languages for describing colors specified by a triplet (r, g, b) of values between 0 and 255, corresponding to the primary colors *red*, *blue* and *green*. Since the number of possible values for the components is $256 = 16^2$, only 2 digits are required to represent these in the hexadecimal notation. Hexadecimal numbers are also employed in IPv6 addresses.

1.1.1 Conversion between binary and decimal formats

Obtaining the decimal representation of a binary number can be achieved from (1.1), using the decimal representations of the powers of 2. Since all the positive and negative powers of 2 have a finite decimal representation, any real number with a finite representation in base 2 has a finite representation also in base 10. For example, $(0.01)_2 = (0.25)_{10}$ and $(0.111)_2 = (0.875)_{10}$.

Example 1.1 (Converting a binary number to decimal notation). Let us calculate the decimal representation of $x = (0.\overline{10})_2$, where the horizontal line indicates repetition: $x = (0.101010\dots)_2$. By definition, it holds that

$$x = \sum_{k=0}^{\infty} a_k 2^{-k},$$

where $a_k = 0$ if k is even and 1 otherwise. Thus, the series may be rewritten as

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

We recognize on the right-hand side a geometric series with common ratio $r = 2^{-2} = \frac{1}{4}$, and so we obtain

$$x = \frac{1}{2} \left(\frac{1}{1-r} \right) = \frac{2}{3} = (0.\overline{6})_{10}.$$

Obtaining the binary representation of a decimal number is more difficult, because negative powers of 10 have *infinite* binary representations, as Exercise 1.4 demonstrates. There is, however, a simple procedure to perform the conversion, which we present for the specific case of a real number x with decimal representation of the form $x = (0.a_1 \dots a_n)_{10}$. In this setting, the bits $(b_1, b_2, \dots)$ in the binary representation of $x = (0.b_1 b_2 b_2 \dots)_2$ may be obtained as follows:

Algorithm 1 Conversion of a number to binary format

```
1:  i ← 1
2:  while x ≠ 0 do
3:      x ← 2x
4:      if x ≥ 1 then
5:          b_i ← 1
6:      else
7:          b_i ← 0
8:      end if
9:      x ← x − b_i
10:     i ← i + 1
11: end while
```

Example 1.2 (Converting a decimal number to binary notation). Let us calculate the binary representation of $x = \frac{1}{3} = (0.\overline{3})_{10}$. We apply Algorithm 1 and collate the values of i and x obtained at the beginning of each iteration, i.e. just before Line 3, in the table below.

i	x	Result
1	$\frac{1}{3}$	0.**0**000...
2	$\frac{2}{3}$	0.0**1**00...
3	$\frac{1}{3}$	0.00**0**0...

Since x in the last row is again $\frac{1}{3}$, successive bits alternate between 0 and 1, and the binary representation of x is given by $(0.\overline{01})_2$. This is not surprising since $2x = (0.66)_{10} = (0.\overline{10})_2$, as we saw in Example 1.1.

1.1.2 Exercises

Exercise 1.1. *Show that if a number $x \in \mathbf{R}$ admits a finite representation (1.1) in base β, then it also admits an infinite representation in the same base.* ***Hint:*** *You may have learned before that $(0.\overline{9})_{10} = 1$.*

Exercise 1.2. *How many digits does it take to represent all the integers from 0 to $10^{10} - 1$ in decimal and binary format? What about the hexadecimal format?*

Exercise 1.3. *Find the decimal representation of $(0.000\overline{1100})_2$.*

Exercise 1.4. *Find the binary representation of $(0.1)_{10}$.*

Exercise 1.5. *Implement Algorithm 1 on a computer and verify that it works. Your function should take as argument an array of integers containing the digits after the decimal point; that is, an array of the form* `[a_1, ..., a_n]`.

Exercise 1.6. *As mentioned above, Algorithm 1 works only for decimal numbers of the specific form $x = (0.a_1 \dots a_n)_{10}$. Find and implement a similar algorithm for integer numbers. More precisely, write a function that takes an integer n as argument and returns an array containing the bits of the binary expansion $(b_m \dots b_0)_2$ of n, from the least significant b_0 to the most significant b_m. That is to say, your code should return* `[b_0, b_1, ...]`.

```
function to_binary(n)
    # Your code comes here
end

# Check that it works
number = 123456789
bits = to_binary(number)
pows2 = 2 .^ range(0, length(bits) - 1)
@assert sum(bits'pows2) == number
```

1.2 Set of values representable in floating point formats

We mentioned in the introduction that, because of memory limitations, only a subset of the real numbers can be stored exactly in a computer. Nowadays, the vast majority of programming languages and software comply with the IEEE 754 standard, which requires that the set of representable numbers be of the form

$$\mathbf{F}(p, E_{\min}, E_{\max}) = \Big\{(-1)^s 2^E (b_0.b_1 b_2 \dots b_{p-1})_2 : \\ s \in \{0,1\}, b_i \in \{0,1\} \text{ and } E_{\min} \le E \le E_{\max}\Big\}. \tag{1.2}$$

In addition to these, floating number formats provide the special entities `Inf`, `-Inf` and `NaN`, the latter being an abbreviation for *Not a Number*. Three parameters appear in the set definition (1.2). The parameter $p \in \mathbf{N}_{>0}$ is the number of significant bits (also called the precision),

and $(E_{\min}, E_{\max}) \in \mathbf{Z}^2$ are respectively the minimum and maximum exponents. From the precision, the *machine epsilon* is defined as $\varepsilon_M = 2^{-p-1}$; its significance is discussed in Section 1.2.2.

For a number $x \in \mathbf{F}(p, E_{\min}, E_{\max})$, s is called the *sign*, E is the *exponent* and $b_0.b_1b_2 \dots b_{p-1}$ is the *significand*. The latter can be divided into a *leading bit* b_0 and the *fraction* $b_1b_2 \dots b_{p-1}$, to the right of the binary point. The most widely used floating point formats are the *single* and *double precision* formats, which are called respectively `Float32` and `Float64` in Julia. Their parameters, together with those of the lesser-known half-precision format, are summarized in Table 1.1. In the rest of this section we use the shorthand notation $\mathbf{F}_{16}$, $\mathbf{F}_{32}$ and $\mathbf{F}_{64}$. Note that $\mathbf{F}_{16} \subset \mathbf{F}_{32} \subset \mathbf{F}_{64}$.

	Half precision	Single precision	Double precision
p	11	24	53
$E_{\min}$	-14	-126	-1022
$E_{\max}$	15	127	1023

Table 1.1: Floating point formats. The first column corresponds to the *half-precision* format. This format, which is available through the `Float16` type in Julia, is more recent than the single and double precision formats. It was introduced in the 2008 revision to the IEEE 754 standard of 1985, a revision known as IEEE 754-2008.

Remark 1.1. Some definitions, notably that in [9, Section 2.5.2], include a general base β instead of the base 2 as an additional parameter in the definition of the number format (1.2). Since the binary format ($\beta = 2$) is always employed in practice, we focus on this case for simplicity in most of this chapter.

Remark 1.2. Given a real number $x \in \mathbf{F}(p, E_{\min}, E_{\max})$, the exponent E and significand are generally not uniquely defined. For example, the number $2.0 \in \mathbf{F}_{64}$ may be expressed as $(-1)^0 2^1 (1.00\dots00)_2$ or, equivalently, as $(-1)^0 2^2 (0.100\dots00)_2$.

In `Julia`, non-integer numbers are interpreted as `Float64` by default, which can be verified by using the `typeof` function. For example, the instruction "`a = 0.1`" is equivalent to "`a = Float64(0.1)`". In order to define a number of type `Float32`, the suffix `f0` must be appended to the decimal expansion. For instance, the instruction "`a = 4.0f0`" defines a floating point number `a` of type `Float32`; it is equivalent to writing "`a = Float32(4.0)`".

1.2.1 Denormalized floating point numbers

We can decompose the set $\mathbf{F}(p, E_{\min}, E_{\max})$ in two disjoint parts:

$$\begin{aligned}\mathbf{F}(p, E_{\min}, E_{\max}) = \Big\{ &(-1)^s 2^E (1.b_1b_2 \dots b_{p-1})_2 : \\ &s \in \{0,1\}, b_i \in \{0,1\} \text{ and } E_{\min} \le E \le E_{\max} \Big\} \\ \cup \Big\{ &(-1)^s 2^{E_{\min}} (0.b_1b_2 \dots b_{p-1})_2 : s \in \{0,1\}, b_i \in \{0,1\} \Big\}.\end{aligned}$$

The numbers in the second set are called *subnormal* or *denormalized*.

1.2.2 Relative error and machine epsilon

Let x be a nonzero real number and $\widehat{x}$ be an approximation. We define the absolute and relative errors of the approximation as follows.

Definition 1.1 (Absolute and relative error)**.** The absolute error is given by $|x - \widehat{x}|$, whereas the relative error is

$$\frac{|x - \widehat{x}|}{|x|}$$

The following result establishes a link between the machine ε_M and the relative error between a real number and the closest member of a floating point format.

Proposition 1.1. *Let $x_{\min}$ and $x_{\max}$ denote the smallest and largest* non-denormalized *positive numbers in a format $F = \mathbf{F}(p, E_{\min}, E_{\max})$. If $x \in [-x_{\max}, -x_{\min}] \cup [x_{\min}, x_{\max}]$, then*

$$\min_{\widehat{x} \in F} \frac{|x - \widehat{x}|}{|x|} \leq \frac{1}{2} 2^{-(p-1)} = \frac{1}{2} \varepsilon_M. \tag{1.3}$$

Proof. For simplicity, we also assume that $x > 0$. Let us introduce $n = \lfloor \log_2(x) \rfloor$ and $y := 2^{-n}x$. Since $y \in [1, 2)$, it has a binary representation of the form $(1.b_1 b_2 \dots)_2$, where the bits after the binary point are not all equal to 1 ad infinitum. Thus $x = 2^n (1.b_1 b_2 \dots)_2$, and from the assumption that $x_{\min} \leq x \leq x_{\max}$ we deduce hat $E_{\min} \leq n \leq E_{\max}$. We now define the number $x_- \in F$ by truncating the binary expansion of x as follows:

$$x_- = 2^n (1.b_1 \dots b_{p-1})_2.$$

The distance between x_- and its successor in F, which we denote by x_+, is given by 2^{n-p+1}. Consequently, it holds that

$$(x_+ - x) + (x - x_-) = x_+ - x_- = 2^{n-p+1}.$$

Since both summands on the left-hand side are positive, this implies that either $x_+ - x$ or $x - x_-$ is bounded from above by $\frac{1}{2} 2^{n-p+1} \leq \frac{1}{2} 2^{-p+1} x$, which concludes the proof. □

The machine epsilon, which was defined as $\varepsilon_M = 2^{-(p-1)}$, coincides with the maximum relative spacing between a non-denormalized floating point number x and its successor in the floating point format, defined as the smallest number in the format that is strictly larger than x.

Figure 1.1 depicts the density of double-precision floating point numbers, i.e. the number of $\mathbf{F}_{64}$ members per unit on the real line. The figure shows that the density decreases as the absolute value of x increases. We also notice that the density is piecewise constant with discontinuities at powers of 2. Figure 1.2 illustrates the relative spacing between successive floating point numbers. Although the absolute spacing increases with the absolute value of x, the relative spacing oscillates between $\frac{1}{2}\varepsilon_M$ and ε_M.

The picture of the relative spacing between successive floating point numbers looks quite different for denormalized numbers. This is illustrated in Figure 1.3, which shows that the

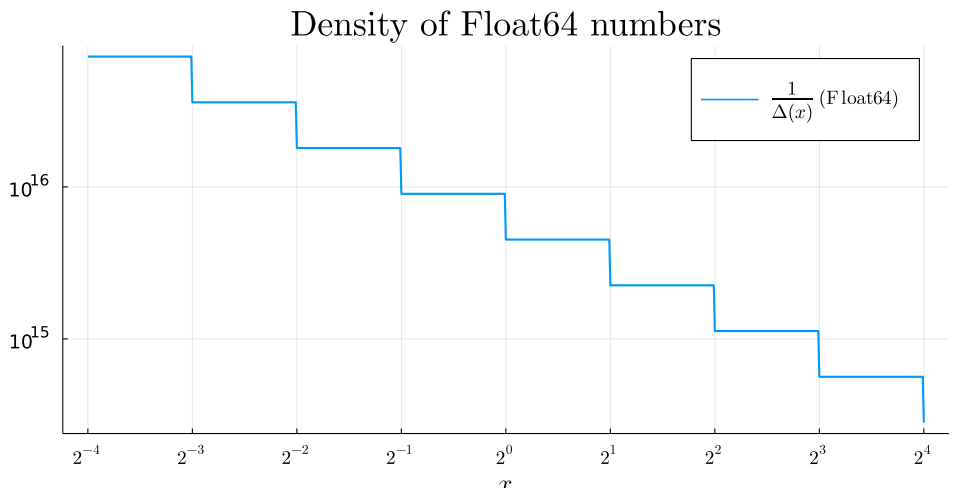


Figure 1.1: Density of the double-precision floating point numbers, measured here as $1/\Delta(x)$ where, for $x \in \mathbf{F}_{64}$, $\Delta(x)$ denotes the distance between x and its successor in $\mathbf{F}_{64}$.

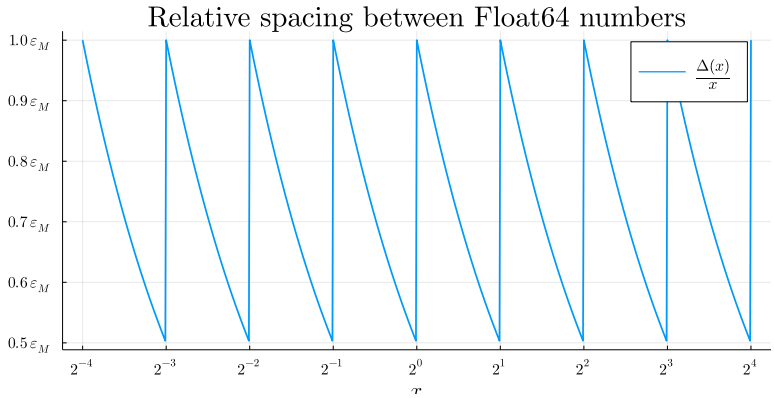


Figure 1.2: Relative spacing between successive double-precision floating point numbers in the "normal range". The relative spacing oscillates between $\frac{1}{2}\varepsilon_M$ and ε_M.

relative spacing increases beyond the machine epsilon in the denormalized range. Fortunately, in the usual $\mathbf{F}_{32}$ and $\mathbf{F}_{64}$ formats, the transition between non-denormalized and denormalized numbers occurs at such a small value that it rarely needs worrying about.

Remark 1.3. In Julia, the machine epsilon can be obtained using the `eps` function. For example, the instruction `eps(Float16)` returns ε_M for the half-precision format.

1.2.3 Exercises

Exercise 1.7. *Write down the values of the smallest and largest, in absolute value, positive real numbers representable in the* $\mathbf{F}_{32}$ *and* $\mathbf{F}_{64}$ *formats.*

Exercise 1.8 (Relative error and machine epsilon)**.** *Prove that the inequality* (1.3) *is sharp. To this end, find* $x \in \mathbf{R}$ *such that the inequality is an equality.*

Exercise 1.9 (Cardinality of the set of floating point numbers)**.** *Show that, if* $E_{\max} \geq E_{\min}$,

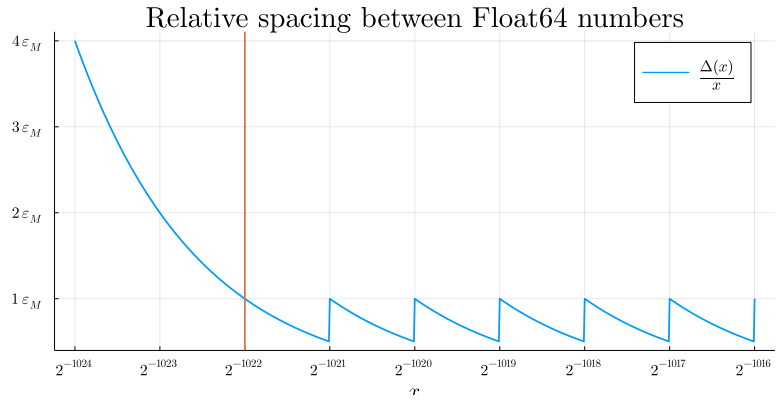


Figure 1.3: Relative spacing between successive double-precision floating point numbers, over a range which includes denormalized number. The vertical red line indicates the transition from denormalized to non-denormalized numbers.

then $\mathbf{F}(p, E_{\min}, E_{\max})$ *contains exactly*

$$(E_{\max} - E_{\min})2^p + 2^{p+1} - 1$$

distinct real numbers. *(In particular, the special values* `Inf`*,* `-Inf` *and* `NaN` *are not counted.)* ***Hint:*** *Count first the numbers with* $E > E_{\min}$ *and then those with* $E = E_{\min}$*.*

1.3 Arithmetic operations between floating point formats

Now that we have presented the set of values representable on a computer, we attempt in this section to understand precisely how arithmetic operations between floating point formats are performed. The key mechanism governing arithmetic operations on a computer is that of *rounding*, the action of approximating a real number regarded as infinitely precise by a number in a floating point format $\mathbf{F}(p, E_{\min}, E_{\max})$. The IEEE 754 standard stipulates that the default mechanism for rounding a real number x, called *round to nearest*, should behave as follows:

- **Standard case**: The number x is rounded to the *nearest representable number*, if this number is unique.
- **Edge case**: When there are two equally near representable numbers in the floating point format, the one with the least significant bit equal to zero is delivered.
- **Infinities**: If the real number x is larger than the largest representable number in the format, that is larger than or equal to $x_{\max} = 2^{E_{\max}}(2 - 2^{-p-1})$, then there are two cases,
  - If $x < 2^{E_{\max}}(2 - 2^{-p})$, then $x_{\max}$ is delivered;
  - Otherwise, the special value `Inf` is delivered.

  In other words, $x_{\max}$ is delivered if it would be delivered by following the rules of the first two bullet points in a different floating point format with the same precision but a larger exponent $E_{\max}$. A similar rule applies for large negative numbers.

When a binary arithmetic operation $(+,\ -,\ \times,\ /)$ is performed on floating point numbers in format $\mathbf{F}$, the result delivered by the computer is obtained by rounding the exact result of the operation according to the rules given above. In other words, the arithmetic operation is performed as if the computer first calculated an intermediate exact result, and then rounded this intermediate result in order to provide a final result in $\mathbf{F}$.

Mathematically, arithmetic operations between floating point numbers in a given format $\mathbf{F}$ may be formalized by introducing the rounding operator $\mathrm{fl} : \mathbf{R} \to \mathbf{F}$ and by defining, for any binary operation $\circ \in \{+, -, \times, /\}$, the corresponding machine operation

$$\widehat{\circ}\colon \mathbf{F} \times \mathbf{F} \to \mathbf{F}; (x, y) \mapsto \mathrm{fl}(x \circ y).$$

We defined this operator for arguments in the same floating point format $\mathbf{F}$. If the arguments of a binary arithmetic operation are of different types, the format of the end result, known as the *destination format*, depends on that of the arguments: as a rule of thumb, it is given by the most precise among the formats of the arguments. In addition, recall that a floating point literal whose format is not explicitly specified is rounded to double-precision format and so, for example, the addition `0.1 + 0.1` produces the result $\mathrm{fl}_{64}\big(\mathrm{fl}_{64}(0.1) + \mathrm{fl}_{64}(0.1)\big)$, where fl_{64} is the rounding operator to the double-precision format.

Example 1.3. Using the `typeof` function, we check that the floating point literal `1.0` is indeed interpreted as a double-precision number:

```
julia> a = 1.0; typeof(a)
Float64
```

When two numbers in different floating point formats are passed to a binary operation, the result is in the more precise format.

```
julia> typeof(Float16(1) + Float32(1))
Float32


julia> typeof(Float32(1) + Float64(1))
Float64
```

If a mathematical expression contains several binary arithmetic operations to be performed in succession, the result of each intermediate calculation is stored in a floating point format dictated by the formats of its argument, and this floating point number is employed in the next binary operation. A consequence of this mechanism is that the machine operands $\widehat{+}$ and $\widehat{*}$ are generally *not associative*. For example, in general

$$(x \mathbin{\widehat{+}} y) \mathbin{\widehat{+}} z \neq x \mathbin{\widehat{+}} (y \mathbin{\widehat{+}} z)$$

Example 1.4. Let $x = 1$ and $y = 3 \times 2^{-13}$. Both of these numbers belong to $\mathbf{F}_{16}$ and, denoting by $\widehat{+}$ machine addition in $\mathbf{F}_{16}$, we have

$$(x \,\widehat{+}\, y) \,\widehat{+}\, y = 1 \tag{1.4}$$

but

$$x \,\widehat{+}\, (y \,\widehat{+}\, y) = 1 + 2^{-10}. \tag{1.5}$$

To explain this somewhat surprising result, we begin by writing the normalized representations of x and y in the $\mathbf{F}_{16}$ format:

$$x = (-1)^0 \times 2^0 \times (\texttt{1.0000000000})_2$$
$$y = (-1)^0 \times 2^{-12} \times (\texttt{1.1000000000})_2.$$

The exact result of the addition $x + y$ is given by $r = 1 + 3 \times 2^{-13}$, which in binary notation is

$$r = (1.\underbrace{00000000000}_{11 \text{ zeros}}11)_2.$$

Since the length of the significand in the half-precision ($\mathbf{F}_{16}$) format is only $p = 11$, this number is not part of $\mathbf{F}_{16}$. The result of the machine addition $\widehat{+}$ is therefore obtained by rounding r to the nearest member of $\mathbf{F}_{16}$, which is 1. This reasoning can then be repeated in order to conclude that, indeed,

$$(x \,\widehat{+}\, y) \,\widehat{+}\, y = x \,\widehat{+}\, y = 1.$$

In order to explain the result of (1.5), note that the exact result of the addition $y + y$ is $r = 3 \times 2^{-12}$, which belongs to the floating point format, so it also holds that $y \,\widehat{+}\, y = 3 \times 2^{-12}$. Therefore,

$$x \,\widehat{+}\, (y \,\widehat{+}\, y) = 1 \,\widehat{+}\, 3 \times 2^{-12} = \mathrm{fl}_{16}(1 + 3 \times 2^{-12}).$$

The argument of the $\mathbf{F}_{16}$ rounding operator does not belong to $\mathbf{F}_{16}$, since its binary representation is given by

$$(1.\underbrace{0000000000}_{10 \text{ zeros}}11)_2.$$

This time the nearest member of $\mathbf{F}_{16}$ is given by $1 + 2^{-10}$.

When a numerical computation unexpectedly returns `Inf` or `Inf`, we say that an *overflow error* occurred. Similarly, *underflow* occurs when a number is smaller than the smallest representable number in a floating point format.

1.3.1 Exercises

Exercise 1.10. *Calculate the machine epsilon ε_{16} for the $\mathbf{F}_{16}$ format. Write the results of the arithmetic operations $1 \,\widehat{+}\, \varepsilon_{16}$ and $1 \,\widehat{-}\, \varepsilon_{16}$ in $\mathbf{F}_{16}$ normalized representation.*

Exercise 1.11 (Catastrophic cancellation)**.** *Let ε_{16} be the machine epsilon for the $\mathbf{F}_{16}$*

format, and define $y = \frac{4}{3}\varepsilon_{16}$. *What is the relative error between* $\Delta = (1+y)-1$, *and the machine approximation* $\widehat{\Delta} = (1 \mathbin{\widehat{+}} y) \mathbin{\widehat{-}} 1$?

Exercise 1.12 (Numerical differentiation). *Let* $f(x) = \exp(x)$. *By definition, the derivative of* f *at* 0 *is*

$$f'(0) = \lim_{\delta \to 0} \left(\frac{f(\delta) - f(0)}{\delta} \right).$$

It is natural to use the expression within brackets on the right-hand side with a small but nonzero δ *as an approximation for* $f'(0)$. *Implement this approach using double-precision numbers and the same values for* δ *as in the table below. Explain the results you obtain.*

δ	$\frac{\varepsilon_{64}}{4}$	$\frac{\varepsilon_{64}}{2}$	ε_{64}
$f'(0)$	*0*	*2*	*1*

Exercise 1.13 (Avoiding overflow). *Write a code to calculate the weighted average*

$$S := \frac{\sum_{j=0}^{J} w_j j}{\sum_{j=0}^{J} w_j}, \qquad w_j = \exp(j), \qquad J = 1000.$$

Exercise 1.14 (Calculating the sample variance). *Assume that* $(x_n)_{1 \le n \le N}$, *with* $N = 10^6$, *are independent random variables distributed according to the uniform distribution* $\mathcal{U}(L, L+1)$. *That is, each* x_n *takes a random value uniformly distributed between* L *and* $L+1$ *where* $L = 10^9$. *In Julia, these samples can be generated with the following lines of code:*

```
N, L = 10^6, 10^9
x = L .+ rand(N)
```

It is well know that the variance of $x_n \in \mathcal{U}(L, L+1)$ *is given by* $\sigma^2 = \frac{1}{12}$. *Numerically, the variance can be estimated from the* sample variance*:*

$$s^2 = \frac{1}{N-1} \left(\left(\sum_{n=1}^{N} x_n^2 \right) - N\bar{x}^2 \right), \qquad \bar{x} = \frac{1}{N} \sum_{n=1}^{N} x_n. \tag{1.6}$$

Write a computer code to calculate s^2 *with the best possible accuracy. Can you find a formula that enables better accuracy than* (1.6)*?*

Remark 1.4. In order to estimate the true value of s^2 for your samples, you can use the `BigFloat` format, to which the array x can be converted by using the instruction `x = BigFloat.(x)`.

Exercise 1.15. *Euler proved that*

$$\frac{\pi^2}{6} = \lim_{N \to \infty} \sum_{n=1}^{N} \frac{1}{n^2}.$$

Using the default `Float64` *format, estimate the error obtained when the series on the right-hand side is truncated after* 10^{10} *terms. Can you rearrange the sum for best accuracy?*

Exercise 1.16. *Let x and y be positive real numbers in the interval $[2^{-10}, 2^{10}]$ (so that we do not need to worry about denormalized numbers, assuming we are working in single or double precision), and let us define the machine addition operator $\widehat{+}$ for arguments in real numbers as*

$$\widehat{+} : \mathbf{R} \times \mathbf{R} \to \mathbf{R}; (x, y) \mapsto \mathrm{fl}\big(\mathrm{fl}(x) + \mathrm{fl}(y)\big).$$

Prove the following bound on the relative error between the sum $x + y$ and its machine approximation $x \,\widehat{+}\, y$:

$$\frac{\big|(x+y) - (x \,\widehat{+}\, y)\big|}{|x+y|} \leq \frac{\varepsilon_M}{2}\left(2 + \frac{\varepsilon_M}{2}\right).$$

Hint: *decompose the numerator as*

$$(x+y) - (x \,\widehat{+}\, y) = \big(x - \mathrm{fl}(x)\big) + \big(y - \mathrm{fl}(y)\big) + \big(\mathrm{fl}(x) + \mathrm{fl}(y) - (x \widehat{+} y)\big),$$

and then use Proposition 1.1.

Exercise 1.17. *Is* `Float32(0.1) * Float32(10) == 1` *equal to* `true` *or* `false` *given the default rounding rule defined by the IEEE standard? Explain.*

Solution. By default, real numbers are rounded to the nearest floating point number. This can be checked in Julia with the command `rounding(Float32)`, which prints the default rounding mode. The exact binary representation of the real number $x = 0.1$ is

$$\begin{aligned} x &= (0.000\overline{1100})_2 \\ &= 2^{-4} \times (\underbrace{1.10011001100110011001100}_{\text{24 bits}}\,\overline{1100})_2 \end{aligned}$$

The first task is to determine the member of $\mathbf{F}_{32}$ that is nearest x. We have

$$\begin{aligned} x^- &= \max\big\{x : x \in \mathbf{F}_{32} \text{ and } x \leq \sqrt{2}\big\} = 2^{-4} \times (1.10011001100110011001100)_2 \\ x^+ &= \min\big\{x : x \in \mathbf{F}_{32} \text{ and } x \geq \sqrt{2}\big\} = 2^{-4} \times (1.10011001100110011001101)_2. \end{aligned}$$

Since the number $(0.\overline{1100})_2$ is closer to 1 than to 0, the number x is closer to x^+ than to x^-. Therefore, the number obtained when writing `Float32(0.1)` is x^+. To conclude the exercise, we need to calculate $\mathrm{fl}\big(10 \times x^+\big)$, and to this end we first write the exact binary representation of the real number $10 \times x^+ = (1010)_2 \times x^+$. We have

$$\begin{aligned} (1010)_2 \times x^+ = (1000)_2 \times x^+ + (10)_2 \times x^+ &= 2^{-4} \times \ (1100.11001100110011001101)_2 \\ &\quad + 2^{-4} \times \quad (11.00110011001100110011101)_2 \\ &= 2^{-4} \times (\underbrace{10000.00000000000000000000}_{\text{24 bits}}001)_2. \end{aligned}$$

This can be checked in Julia by writing `bitstring(Float32(0.1) * Float64(10.0))`. Clearly, when rounding to the nearest $\mathbf{F}_{32}$ number, the number $2^{-4}(10000)_2 = 1$ is obtained.

Remark 1.5. It should not be inferred from Exercise 1.17 that `Float32(1/i) * i` is always exact in floating point arithmetic. For example `Float32(1/41) * 41` does not evaluate to `1`, and neither do `Float16(1/11) * 11` and `Float64(1/49) * 49`.

Exercise 1.18. *Explain why* `Float32(sqrt(2))^2 - 2` *is not zero in Julia.*

Solution. The exact binary representation of $x := \sqrt{2}$ is

$$x = (\underbrace{1.01101010000010011110011}_{24 \text{ bits}} 001100\ldots)_2.$$

The first task is to determine the member of $\mathbf{F}_{32}$ that is nearest x. We have

$$x^- = \max\{x : x \in \mathbf{F}_{32} \text{ and } x \le \sqrt{2}\} = (\underbrace{1.01101010000010011110011}_{24 \text{ bits}})_2$$
$$x^+ = \min\{x : x \in \mathbf{F}_{32} \text{ and } x \ge \sqrt{2}\} = (\underbrace{1.01101010000010011110100}_{24 \text{ bits}})_2,$$

and we calculate

$$x - x^- = 2^{-24}(0.01100\ldots)_2,$$
$$x^+ - x = 2^{-21}\big(1 - (0.11001100\ldots)_2\big) \ge 2^{-21}\big(1 - (0.11001101)_2\big) = 2^{-21}\,(0.00110011)_2\,.$$

We deduce that $x - x^- \le x^+ - x$, and so $\mathrm{fl}(x) = x^-$. To conclude the exercise, we need to show that $\mathrm{fl}\big((x^-)^2\big)$ is not equal to 2. The exact binary expansion of $(x^-)^2$ is

$$(x^-)^2 = (\underbrace{1.11111111111111111111111}_{24 \text{ bits}} 011011)_2.$$

The member of $\mathbf{F}_{32}$ nearest this number is

$$(1.11111111111111111111111)_2 = 2 - 2^{-23},$$

which is precisely the result returned by Julia.

1.4 Encoding of floating point numbers

Once a number format is specified through parameters $(p, E_{\min}, E_{\max})$, the choice of encoding, i.e. the machine representation of numbers in this format, has no bearing on the magnitude and propagation of round-off errors. Studying encodings is, therefore, not essential for our purposes in this course, but we opted to cover the topic anyway in the hope that it will help the students build intuition on floating point numbers. We focus mainly on the single precision format, but the following discussion applies *mutatis mutandis* to the double and half-precision formats. The material in this section is for information purposes only.

We already mentioned in Remark 1.2 that a number in a floating point format may have several representations. On a computer, however, a floating point number is always stored in the same manner (except for the number 0, see Remark 1.7). The values of the exponent and

significand which are selected by the computer, in the case where there are several possible choices, are determined from the following rules:

- Either $E > E_{\min}$ and $b_0 = 1$;
- Or $E = E_{\min}$, in which case the leading bit may be 0.

The following result proves that these rules define the exponent and significand uniquely.

Proposition 1.2. *Assume that*

$$(-1)^s(2^E b_0.b_1 \dots b_{p-1})_2 = (-1)^{\widetilde{s}}(2^{\widetilde{E}}\widetilde{b}_0.\widetilde{b}_1 \dots \widetilde{b}_{p-1})_2, \tag{1.7}$$

where the parameter sets $(s, E, b_0, \dots b_{p-1})$ *and* $(\widetilde{s}, \widetilde{E}, \widetilde{b}_0, \dots, \widetilde{b}_{p-1})$ *both satisfy the above rule. Then* $E = \widetilde{E}$ *and* $b_i = \widetilde{b}_i$ *for* $i \in \{0, \dots, p-1\}$.

Proof. We show that $E = \widetilde{E}$, after which the equality of significands follows trivially. Let us assume for contradiction that $E > \widetilde{E}$ and denote the left and right-hand sides of (1.7) by x and $\widetilde{x}$, respectively. Then $E > E_{\min}$, implying that $b_0 = 1$ and so $2^E \leq |x| < 2^{E+1}$. On the other hand, it holds that $|\widetilde{x}| < 2^{\widetilde{E}+1}$ regardless of whether $\widetilde{E} = E_{\min}$ or not. Since $E \geq \widetilde{E} + 1$ by assumption, we deduce that $|\widetilde{x}| < 2^E \leq |x|$, which contradicts the equality $x = \widetilde{x}$. □

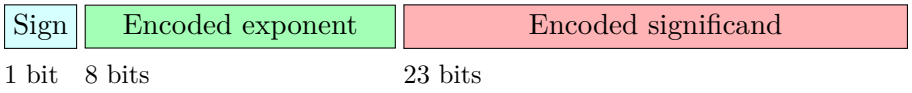
Now that we explained how a unique set of parameters (sign, exponent, significand) can be assigned to any floating point number, we describe how these parameters are stored on the computer in practice? As their names suggest, the `Float16`, `Float32` and `Float64` formats use 16, 32 and 64 bits of memory, respectively. A naive approach for encoding these number formats would be to store the full binary representations of the sign, exponent and significand.

For the `Float32` format, this approach would requires 1 bit for the sign, 8 bits to cover the 254 possible values of the exponent, and 24 bits for the significand, i.e. for storing $b_0, \dots, b_{p-1}$. This leads to a total number of 33 bits, which is one more than is available, and this is without the special values `NaN`, `Inf` and `-Inf`. So how are numbers in the $\mathbf{F}_{32}$ format actually stored? To answer this question, we begin with two observations:

- If $E > E_{\min}$, then necessarily $b_0 = 1$ in the unique representation of the significand. Consequently, the leading bit need not be explicitly specified in the case; it is said to be *implicit*. As a consequence, we will see that $p - 1$ instead of p bits are in fact sufficient for the significand.
- In the $\mathbf{F}_{32}$ format, 8 bits at minimum need to be reserved for the exponent, which enables the representation of $2^8 = 256$ different values, but there are only 254 possible values for the exponent. This suggests that $256 - 254 = 2$ combinations of the 8 bits can be exploited in order to represent the special values `Inf`, `-Inf` and `NaN`.

Simplifying a little, we may view single precision floating point number as an array of 32

bits as illustrated below:



According to the IEEE 754 standard, the first bit is the sign s, the next 8 bits $e_0e_1 \dots e_6e_7$ encode the exponent, and the last 23 bits $b_1b_2 \dots b_{p-2}b_{p-1}$ encode the significand. Let us introduce the integer number $e = (e_0e_1 \dots e_6e_7)_2$; that is to say, $0 \leq e \leq 2^8 - 1$ is the integer number whose binary representation is given by $e_0e_1 \dots e_6e_7$. One may determine the exponent and significand of a floating point number from the following rules.

- **Denormalized numbers**: If $e = 0$, then the implicit leading bit b_0 is zero, the fraction is $b_1b_2 \dots b_{p-2}b_{p-1}$, and the exponent is $E = E_{\min}$. In other words, using the notation of Section 1.2, we have $x = (-1)^s 2^{E_{\min}} (0.b_1b_2 \dots b_{p-2}b_{p-1})_2$. In particular, if $b_1b_2 \dots b_{p-2}b_{p-1} = 00 \dots 00$, then it holds that $x = 0$.

- **Non-denormalized numbers**: If $0 < e < 255$, then the implicit leading bit b_0 of the significand is 1 and the fraction is given by $b_1b_2 \dots b_{p-2}b_{p-1}$. The exponent is given by

$$E = e - \text{bias} = E_{\min} + e - 1.$$

  where the exponent bias for the single and double precision formats are given in Table 1.2. In this case $x = (-1)^s 2^{e-\text{bias}} 1.b_1b_2 \dots b_{p-2}b_{p-1}$. Notice that $E = E_{\min}$ if $e = 1$, as in the case of subnormal numbers.

- **Infinities**: If $e = 255$ and $b_1b_2 \dots b_{p-2}b_{p-1} = 00 \dots 00$, then $x =$ `Inf` if $s = 0$ and `-Inf` otherwise.

- **Not a Number**: If $e = 255$ and $b_1b_2 \dots b_{p-2}b_{p-1} \neq 00 \dots 00$, then $x =$ `NaN`. Notice that the special value `NaN` can be encoded in many different manners. These extra degrees of freedom were reserved for passing information on the reason for the occurrence of `NaN`, which is usually an indication that something has gone wrong in the calculation.

	Half precision	Single precision	Double precision
Exponent bias ($-E_{\min} + 1$)	15	127	1023
Exponent encoding (bits)	5	8	11
Significand encoding (bits)	10	23	52

Table 1.2: Encoding parameters for floating point formats

Remark 1.6 (Encoding efficiency). With 32 bits, at most 2^{32} different numbers could in principle be represented. In practice, as we saw in Exercise 1.9, the `Float32` format enables to represent

$$(E_{\max} - E_{\min})2^p + 2^{p+1} - 1 = 253 \times 2^{23} + 2^{25} - 1 = 2^{32} - 2^{24} - 1 \approx 99.6\% \times 2^{32},$$

different real numbers, which is a very good efficiency.

Remark 1.7 (Nonuniqueness of the floating point represention of 0.0). The sign s is clearly unique for any number in a floating point format, except for 0.0, which could in principle be represented as

$$(-1)^0 2^{E_{\min}}(0.00\ldots00)_2 \qquad \text{or} \qquad (-1)^1 2^{E_{\min}}(0.00\ldots00)_2.$$

In practice, both representations of 0.0 are available on most machines, and these behave slightly differently. For example `1/(0.0) = Inf` but `1/(-0.0) = -Inf`.

Exercise 1.19. *Determine the encoding of the following* `Float32` *numbers:*

- $x_1 = 2.0^{E_{\min}}$
- $x_2 = -2.0^{E_{\min}-p-1} = -2.0^{-149}$
- $x_3 = 2.0^{E_{\max}}(2 - 2^{-p+1})$

Check your results using the Julia function `bitstring`.

1.5 Integer formats

The machine representation of integer formats is much simpler than that of floating point numbers. In this short section, we give a few orders of magnitude for common integer formats and briefly discuss overflow issues. Programming languages typically provide integer formats based on 16, 32 and 64 bits. In Julia, these correspond to the types `Int16`, `Int32` and `Int64`, the latter being the default for integer literals.

The most common encoding for integer numbers, which is used in Julia, is known as *two's complement*: a number encoded with p bits as $b_{p-1}b_{p-2}\ldots b_0$ corresponds to

$$x = -b_{p-1}2^{p-1} + \sum_{i=0}^{p-2} b_i 2^i.$$

This encoding enables to represent uniquely all the integers from $N_{\min} = -2^{p-1}$ to $N_{\max} = 2^{p-1} - 1$. In contrast with floating point formats, integer formats do not provide special values like `Inf` and `NaN`. The number delivered by the machine when a calculation exceeds the maximum representable value in the format, called the *overflow behavior*, generally depends on the programming language.

Since overflow behavior of integer numbers is not universal across programming languages, a detailed discussion is of little interest. We only mention that Julia uses a *wraparound* behavior, where $N_{\max} + 1$ silently returns $N_{\min}$ and, similarly, $-N_{\min} - 1$ gives $N_{\max}$; the numbers loop back. This can lead to unexpected results, such as `2^64` evaluating to 0.

1.6 Discussion and bibliography

This chapter is mostly based on the original 1985 IEEE 754 standard [3] and the reference book [9]. A significant revision to the 1985 IEEE standard was published in 2008 [4], adding for example specifications for the half precision and quad precision formats, and a minor revision was published in 2019 [5]. The original IEEE standard and its revisions constitute the authoritative guide on floating point formats. It was intended to be widely disseminated and is written very clearly and concisely, but is not available for free online. Another excellent source for learning about floating point numbers and round-off errors is D. Goldberg's paper "*What every computer scientist should know about floating-point arithmetic*" [2], freely available online.