Chapter 1

Representation of numbers

Since none of the numbers which we take out from logarithmic and trigonometric tables admit of absolute precision, but all are to a certain extent approximate only, the results of all calculations performed by the aid of these numbers can only be approximately true.
It may happen, that in special cases the effect of the errors of the tables is so augmented that we may be obliged to reject a method, otherwise the best, and substitute another in its place.
— CARL F. GAUSS, *Theoria Motus (1809)*[1]

MATLAB*'s creator Dr. Cleve Moler used to advise foreign visitors not to miss the country's two most awesome spectacles: the Grand Canyon, and meetings of IEEE p754.*
— WILLIAM M. KAHAN[2]

The aim of this chapter is to understand how numbers are represented in computers. While the set of real numbers is infinite, computers can only represent and work with a finite subset. Therefore, we need to understand which numbers are representable and how the operations are performed in this set of representable real numbers. This topic has regained importance during the last years with the advent of GPUs and TPUs for scientific computing and machine learning. In particular, TPUs are designed to perform a high volume of low precision computation and one cannot expect high accuracy.

Example 1.1 The following expression for Euler's number e is known from Analysis:

$$e = \lim_{n\to\infty} \left(1 + \frac{1}{n}\right)^n .$$

One therefore expects that $e_n = \left(1 + \frac{1}{n}\right)^n$ yields increasingly good approximations to e as n increases. In *exact* arithmetic this is indeed true. On the computer,

[1] Translated and quoted in Higham (2002).
[2] See `http://www.cs.berkeley.edu/~wkahan/ieee754status/754story.html`.

roundoff error affects the accuracy of computations and the *computed* value $\hat{e}_n$ behaves quite differently:

Python
```
# Approximation of e, numpy package needed to include e
import numpy as np
for i in range(1,16):
    n = 10.0 ** i; en = (1 + 1/n) ** n
    print('10^%2d %20.15f %20.15f' % (i,en,en-np.e))
```

n	Computed $\hat{e}_n$	Error $\hat{e}_n - e$
10^1	2.593742460100002	-0.124539368359044
10^2	2.704813829421529	-0.*0*13467999037517
10^3	2.716923932235520	-0.*00*1357896223525
10^4	2.718145926824356	-0.*000*135901634689
10^5	2.718268237197528	-0.*0000*13591261517
10^6	2.718280469156428	-0.*00000*1359302618
10^7	2.718281693980372	-0.*000000*134478673
10^8	2.718281786395798	-0.*0000000*42063248
10^9	2.718282030814509	0.*000000*202355464
10^{10}	2.718282053234788	0.*000000*224775742
10^{11}	2.718282053357110	0.*000000*224898065
10^{12}	2.718523496037238	0.*000*241667578192
10^{13}	2.716110034086901	-0.*00*2171794372145
10^{14}	2.716110034087023	-0.*00*2171794372023
10^{15}	3.035035206549262	0.316753378090216

Initially, the accuracy gets better as n increases, as expected. However, at around $n = 10^8$, the accuracy stagnates and even *gets worse* when n continues to increase; for $n = 10^{15}$ not even a single (decimal) digit of e is computed correctly. ◇

Example 1.2 From Analysis, we know that the Taylor series for the exponential function converges for every $x \in \mathbb{R}$:

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \dots .$$

In practice, one can of course only compute a partial sum

$$s_i(x) = \sum_{k=0}^{i} \frac{x^k}{k!}.$$

The remainder of the Taylor expansion admits the expression

$$e^x - s_i(x) = \frac{e^{\xi} x^{i+1}}{(i+1)!}$$

for some $\xi \in \mathbb{R}$ with $0 < |\xi| < |x|$. If we choose i such that $|x|^{i+1}/(i+1)! \le \mathsf{tol} \cdot |s_i(x)|$ is satisfied for a (small) chosen tolerance tol then for negative x it holds that

$$|e^x - s_i(x)| \le \frac{|x|^{i+1}}{(i+1)!} \le \mathsf{tol} \cdot |s_i(x)| \approx \mathsf{tol} \cdot e^x.$$

x	Computed $\hat{s}_i(x)$	$\exp(x)$	$\frac{\lvert \exp(x) - \hat{s}_i(x) \rvert}{\exp(x)}$
-20	5.6218844674e-09	2.0611536224e-09	1.727542676201181
-18	1.5385415977e-08	1.5229979745e-08	0.*0*10205938187564
-16	1.1254180496e-07	1.1253517472e-07	0.*0000*58917020257
-14	8.3152907681e-07	8.3152871910e-07	0.*000000*430176956
-12	6.1442133148e-06	6.1442123533e-06	0.*000000*156480737
-10	4.5399929556e-05	4.5399929762e-05	0.*00000000*4544414
-8	3.3546262817e-04	3.3546262790e-04	0.*000000000*788902
-6	2.4787521758e-03	2.4787521767e-03	0.*000000000*333306
-4	1.8315638879e-02	1.8315638889e-02	0.*000000000*530694
-2	1.3533528320e-01	1.3533528324e-01	0.*000000000*273603
0	1.0000000000e+00	1.0000000000e+00	0.*000000000000000*
2	7.3890560954e+00	7.3890560989e+00	0.*000000000*479969
4	5.4598149928e+01	5.4598150033e+01	0.*00000000*1923058
6	4.0342879295e+02	4.0342879349e+02	0.*00000000*1344248
8	2.9809579808e+03	2.9809579870e+03	0.*00000000*2102584
10	2.2026465748e+04	2.2026465795e+04	0.*00000000*2143800
12	1.6275479114e+05	1.6275479142e+05	0.*00000000*1723845
14	1.2026042798e+06	1.2026042842e+06	0.*00000000*3634135
16	8.8861105010e+06	8.8861105205e+06	0.*00000000*2197990
18	6.5659968911e+07	6.5659969137e+07	0.*00000000*3450972
20	4.8516519307e+08	4.8516519541e+08	0.*00000000*4828738

Table 1.1: Numerical approximation of e^x by the truncated Taylor series $s_i(x)$, where i has been chosen such that the relative error is (approximately) bounded by $\mathsf{tol} = 10^{-8}$ in *exact* arithmetic.

In turn, the *relative error* $|e^x - s_i(x)|/|e^x|$ is approximately bounded by tol. For positive x a similar bound can be found by a refined analysis of the remainder (EFY=Exercise For You).

Python

```python
def expeval(x, tol):
    #Approximation of e^x
    s = 1; k = 1
    term = 1
    while (abs(term)>tol*abs(s)):
        term = term * x / k
        s = s + term
        k = k + 1
    return s
```

Table 1.1 shows the results for $\mathsf{tol} = 10^{-8}$. In contrast to the theoretical results, the relative error is above the imposed (approximate) bound $\mathsf{tol} = 10^{-8}$ for very negative x. ◇

One goal of this chapter is to better understand (and avoid if possible) the phenomena observed in Examples 1.1 and 1.2. To achieve this, we first need to discuss the representation and approximation of real numbers on computers.

For more interesting stories of what has gone (terribly) wrong in the world because of roundoff errors, see [4].

1.1 Representation of real numbers

The first step in representing a real number on a computer is to express it in terms of its digits with respect to a base $\beta \geq 2$. The following theorem, which we state without proof, gives the so called normalized representation or scientific notation of a number.

Theorem 1.3 *Given a* **base** $2 \leq \beta \in \mathbb{N}$, *every nonzero* $x \in \mathbb{R}$ *can be represented as*

$$x = \pm\beta^e \left(\frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \frac{d_3}{\beta^3} + \cdots \right) \tag{1.1}$$

with the **digits** $d_1, d_2, \ldots, \in \{0, 1, 2, \ldots, \beta - 1\}$, *where* $d_1 \neq 0$, *and the* **exponent** $e \in \mathbb{Z}$.

The representation (1.1) becomes unique if we additionally require that there is an infinite subset $\mathbb{N}_1 \subset \mathbb{N}$ such that $d_k \neq \beta - 1$ for all $k \in \mathbb{N}_1$. Otherwise, $x = +10^1 \cdot 0.1$ and $x = +10^0 \cdot 0.999\ldots$ would be two different representations of the same number. However, this aspect will be irrelevant for us; on a computer we can only work with a finite number of digits anyway.

System	β	Digits
Decimal	10	$0, 1, 2, \ldots, 9$
Binary	2	$0, 1$
Octal	8	0,1,2,...,7
Hexadecimal	16	$0, 1, 2, \ldots, 9$, A,B,C,D,E,F

Les doits ou les poings*

In daily life we mostly use the decimal system ($\beta = 10$). But there are exceptions, $\beta = 12$ also plays a role (hours, months, dozen / douzaine). Also, some tribes used their toes for counting as well ($\beta = 20$). Some regions of the world also use 4, 8 or 16, see `https://en.wikipedia.org/wiki/Numeral_system`. For obvious reasons, almost every computer operates with a binary system ($\beta = 2$) and the hexadecimal system is only used for representing binary numbers more conveniently. A computer developed in Moscow at the end of the 1950ies was based on the ternary system ($\beta = 3$) but it was not a huge success.

To compute the representation of Theorem 1.3 for given $x \in \mathbb{R} \setminus \{0\}$, the following algorithm can be used:

```
Determine $e \in \mathbb{Z}$ such that $x \in \beta^e \tilde{x}$ with $\beta^{-1} \leq \tilde{x} < 1$. Set $j = 1$.
while $\tilde{x} \neq 0$ do
  Set $x = \beta\tilde{x}$.
  Decompose $x = d_j + \tilde{x}$ such that $d_j \in \mathbb{N}, 0 \leq d_j \leq \beta - 1$ and $0 \leq \tilde{x} < 1$.
  $j \leftarrow j + 1$.
end while
```

The binary system has some unexpected consequences. As we will see below, the decimal number 0.2 *cannot* be represented by a finite binary fraction, that is, the algorithm above does not terminate. To avoid this effect, some specialized applications in the finance industry use the decimal system. In most cases, this is emulated by software and, consequently, quite slow. Hardware processors, which directly support decimal operations, are rare; an example was the IBM Power6 processor.

Theorem 1.4 *Consider a nonzero rational number $x = p/q$, where $p, q \in \mathbb{Z}$ have no common divisor. Then x has a finite representation in base $\beta \geq 2$ if and only if each of the prime factors of the denominator q divides β.*

Example 1.5 In base $\beta = 10$, a rational number has a finite representation if and only if it can be written as $\pm\frac{p}{2^n 5^m}$ for some $p, n, m \in \mathbb{N}$. This corresponds to the known fact that in base 10, if the divisor has a factor that is neither 2 nor 5, then the Euclidean division does not finish and becomes periodic.

In base $\beta = 2$ instead, to have a finite representation, a number needs to be written as $\pm\frac{p}{2^n}$ for some $p, n \in \mathbb{N}$. For instance, $x = \frac{1}{5} = (0.2)_{10}$ does not have a finite representation in base 2. This can be checked by applying the algorithm above:

$$\begin{aligned}
x &= 2^{-2}0.8 && \Rightarrow e = -2, \tilde{x} = 0.8 \\
x &= 2\tilde{x} = 1.6 && \\
x &= 1 + 0.6 && \Rightarrow d_1 = 1, \tilde{x} = 0.6 \\
x &= 2\tilde{x} = 1.2 && \Rightarrow d_2 = 1, \tilde{x} = 0.2
\end{aligned}$$

After two loops we are again at 0.2 and the algorithm becomes cyclic. Therefore, $\frac{1}{5} = (0.00110011\ldots)_2 = \left(0.\overline{0011}\right)_2$. ◇

1.2 Floating point numbers on computers

Computers can only store and calculate with finitely many numbers $\mathbb{F} \subset \mathbb{R}$. Apart from specialized devices, such as audio decoding hardware devices, nearly all processors operate with floating point numbers.

Definition 1.6 (Floating point numbers $\mathbb{F}(\beta, t, e_{\min}, e_{\max})$) *Consider $\beta \in \mathbb{N}$, $\beta \geq 2$ (**base**), $t \in \mathbb{N}$ (**length of mantissa/significand**), and $e_{\min} < 0 < e_{\max}$ with $e_{\min}$, $e_{\max} \in \mathbb{Z}$ (**range of exponent**). Then the set $\mathbb{F} = \mathbb{F}(\beta, t, e_{\min}, e_{\max}) \subset \mathbb{R}$ is defined as*

$$\mathbb{F} := \left\{ \pm\beta^e \left(\frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \cdots + \frac{d_t}{\beta^t} \right) : \begin{array}{l} d_1, \ldots, d_t \in \{0, \ldots, \beta - 1\}, \\ d_1 \neq 0, \\ e \in \mathbb{Z},\ e_{\min} \leq e \leq e_{\max}. \end{array} \right\} \cup \{0\}.$$

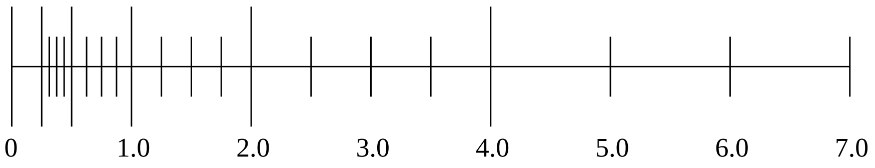
Example 1.7 The floating point number $+2^3(0.1011)_2 = (5.5)_{10}$ belongs to $\mathbb{F}(2, 4, -1, 4)$, but neither to $\mathbb{F}(2, 3, -1, 4)$, nor to $\mathbb{F}(2, 4, -2, 2)$.

Let us consider the set $\mathbb{F} = \mathbb{F}(10, 3, -2, 2)$, that is, $\beta = 10$, $t = 3$, and $-2 \le e \le 2$. Then:

$$\begin{aligned}
23.4 &= +10^2\,(0.234) && \in \mathbb{F} \\
-53.8 &= -10^2\,(0.538) && \in \mathbb{F} \\
3.141 &= +10^1\,(0.3141) && \notin \mathbb{F} \quad (4 > 3 \text{ significant digits}) \\
3.1 &= +10^1\,(0.310) && \in \mathbb{F}
\end{aligned}$$

◇

It is important to keep in mind that the elements of $\mathbb{F}$ are not uniformly distributed on the real line.
Example: Nonnegative floating point numbers for $\beta = 2, t = 3, e_{\min} = -1, e_{\max} = 3$:



Lemma 1.8 *For $\mathbb{F} = \mathbb{F}(\beta, t, e_{\min}, e_{\max})$ one has*

$$x_{\min}(\mathbb{F}) := \min\{x \in \mathbb{F} : x > 0\} = \beta^{e_{\min}-1}, \tag{1.2}$$
$$x_{\max}(\mathbb{F}) := \max\{x \in \mathbb{F}\} \qquad = \beta^{e_{\max}}(1 - \beta^{-t}). \tag{1.3}$$

Proof. In view of Definition 1.6, the verification of (1.2)–(1.3) comes down to determining the range of the mantissa. Consider $x \in \mathbb{F}$ with mantissa $d = \sum_{k=1}^{t} d_k \beta^{-k}$. Because of $d_1 \neq 0$ we have

$$\underset{\substack{\uparrow \\ d_1 \ge 1}}{\beta^{-1} \le d} \quad \underset{\substack{\uparrow \\ d_k \le \beta - 1\,.}}{\le} \sum_{k=1}^{t} \beta^{-k}(\beta - 1) = 1 - \beta^{-t}$$

The smallest positive number is obtained by choosing the mantissa $d_1 = 1, d_2 = 0, d_3 = 0, \ldots$ and the exponent $e = e_{\min}$. The largest number is obtained by choosing the mantissa $d_1 = \beta - 1, d_2 = \beta - 1, d_3 = \beta - 1, \ldots$ and the exponent $e = e_{\max}$. ☐

Lemma 1.9 *The distance between a floating point number x satisfying $x_{\min}(\mathbb{F}) < |x| < x_{\max}(\mathbb{F})$ and the nearest floating point number is* at least $\beta^{-1}\epsilon_M|x|$ *and* at most $\epsilon_M|x|$, *where $\epsilon_M = \beta^{1-t}$ is the distance of* 1 *to the next larger floating point number.*

Proof. Without loss of generality, we may assume that $x \in \mathbb{F}$ is positive and that $e = 0$, which implies that $\beta^{-1} \le x \le 1-\beta^{-t}$. The next larger floating point number

x_+ is obtaining by adding 1 to the last digit d_t of the mantissa, which corresponds to adding β^{-t} to x. In turn, the relative distance between x and x_+ satisifies

$$\frac{x_+ - x}{x} \le \beta^{-t}\beta = \epsilon_M, \qquad \frac{x_+ - x}{x} \ge \frac{\beta^{-t}}{1-\beta^{-t}} \ge \beta^{-t} = \epsilon_M/\beta.$$

In the same manner, analogous bounds are shown for the relative distance between x and the next smaller number x_-. □

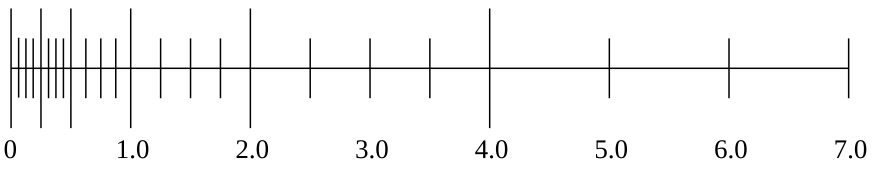
Lemma 1.9 demonstrates that the length of the mantissa determines the relative accuracy of $\mathbb{F}$; the exponent bounds $e_{\min}$, $e_{\max}$ *determine the range of* $\mathbb{F}$ *but not its accuracy.* The number ϵ_M of Lemma 1.9 is usually called *machine precision.*

Subnormal numbers*

As visible in the picture above, there is a “gap” of $\beta^{e_{\min}-1}$ between 0 and the smallest nonnegative floating point number. This gap is caused by the normalization $d_1 \neq 0$ of the mantissa when $e = e_{\min}$. The gap is bridged by adding the so called **subnormal** numbers:

$$\widehat{\mathbb{F}} := \mathbb{F} \cup \left\{ \pm\beta^{e_{\min}} \left(\frac{d_2}{\beta^2} + \cdots + \frac{d_t}{\beta^t} \right) : \ d_2, \ldots, d_t \in \{0, \ldots, \beta - 1\} \right\}, \tag{1.4}$$

where one excludes the case $d_2 = \cdots = d_t = 0$. In other words, one adds numbers with minimal exponent $e_{\min}$ and $d_1 = 0$.
Example: $\widehat{\mathbb{F}}$ for $\beta = 2, t = 3, e_{\min} = -1, e_{\max} = 3$:



Two important remarks:

1. Lemma 1.9 does *not* hold for subnormal numbers, which have a lower relative accuracy.
2. While subnormal numbers are supported by most processors; the computations can slow down significantly because processors are often not optimized to work with subnormal numbers.

In the following, we will ignore subnormal numbers to simplify the discussion.

IEEE standard

IEEE 754 describes the standard for storing and operating with floating point numbers. It also describes the treatment of exceptions ($1/0, \infty \cdot \infty, \infty - \infty, \ldots$). These are the two most widely used formats in base $\beta = 2$:

Name	Size	Mantissa	Exponent	$x_{\min}$	$x_{\max}$
Single precision	32 bits	24 bits	8 bits	10^{-38}	10^{+38}
Double precision	64 bits	53 bits	11 bits	10^{-308}	10^{+308}

Double precision corresponds to $\mathbb{F}(2, 53, -1022, 1023)$ and is the standard on central processing units (CPUs) . One bit of the mantissa is used for the sign. On the other hand, d_1 is *not* saved because the normalization always implies $d_1 = 1$ for $\beta = 2$. In turn, there are $t = 53$ bits for the mantissa. The exponent field is an (unsigned) integer from 0 to 2047; shifted such that it represents the range -1022 to $+1023$ (the fields all 0s and all 1 are reserved for special numbers).

In PYTHON all operations with real numbers are executed in double precision by default. Variables in single precision are generated with the command `numpy.float32()` from the `numpy` package.

PYTHON

```
import sys
sys.float_info.min              # 2.2251e-308
sys.float_info.max              # 1.7977e+308
1 / 0                           # Divide by zero error
3 * float('inf')                # inf
-1 / 0                          # Divide by zero error
0 / 0                           # Divide by zero error
float('inf') - float('inf')     # nan
```

Somewhat surprisingly, and not in accordance with the IEEE 754 standard[3], PYTHON throws an error when a division by zero occurs, instead of returning a signed ∞. To avoid this, one can use `float64` from numpy. For example, `1/numpy.float64(0)` returns `inf` (as it should).

In machine learning, precision is much less important compared to typical applications in scientific computing. On the other hand, speed and memory are key, which makes the bfloat16 format a popular choice that is utilized in many CPUs, GPUs, and AI processors.

Name	Size	Mantissa	Exponent	$x_{\min}$	$x_{\max}$
bfloat16	16 bits	8 bits	8 bits	10^{-38}	10^{+38}
FP8	8 bits	4 bits	4 bits	10^{-2}	240

The use of FP8, which is not standardized yet, is partly responsible for the success of Deepseek.[4] The narrow range of numbers makes it quite difficult to work with.

1.3 Rounding

A real number $x \in \mathbb{R}$ (think of, $\sqrt{2}$ or π) can, in general, not be represented exactly in the computer. The process of approximating x by a number in a floating point

[3]See also `https://wusun.name/blog/2017-12-18-python-zerodiv/` for a discussion.
[4]See `https://verticalserve.medium.com/how-deepseek-optimized-training-fp8-framework-74e3667a2d4a`.

number in $\mathbb{F} = \mathbb{F}(\beta, t, e_{\min}, e_{\max})$ is called **rounding**. More concretely, rounding is a function

$$\mathsf{fl} : r(\mathbb{F}) \to \mathbb{F}$$

which maps $x \in r(\mathbb{F})$ to the[5] nearest element in $\mathbb{F}$. Here, $r(\mathbb{F})$ denotes the range of $\mathbb{F}$:

$$r(\mathbb{F}) := \{x \in \mathbb{R} : x_{\min}(\mathbb{F}) \le |x| \le x_{\max}(\mathbb{F})\};$$

see also Lemma 1.8.

The map fl is not uniquely determined when x is exactly in the middle between two floating point numbers (this *not* a rare event when $\beta = 2$). There are several strategies to break the tie in this situation. According to the IEEE 754 standard, fl chooses the element for which the last digit of the mantissa is even (for $\beta = 2$ this is zero). The following example illustrates this convention.

Example 1.10 Let $\beta = 10$, $t = 3$. Then $\mathsf{fl}(0.9996) = 1.0$, $\mathsf{fl}(0.3345) = 0.334$, $\mathsf{fl}(0.3355) = 0.336$. ◇

The following theorem provides a useful and tight estimate of the relative error of fl.

Theorem 1.11 *For every $x \in r(\mathbb{F})$ there exists $\delta \equiv \delta(x) \in \mathbb{R}$ such that*

$$\mathsf{fl}(x) = x(1+\delta), \quad |\delta| \le u,$$

where $u = u(\mathbb{F}) := \frac{1}{2}\epsilon_M = \frac{1}{2}\beta^{1-t}$.

Proof. Without loss of generality, we may assume $x > 0$. According to Theorem 1.3 we can represent x as follows:

$$x = \mu \cdot \beta^{e-t}, \quad \beta^{t-1} \le \mu < \beta^t.$$

This shows that x is between the adjacent floating point numbers $y_1 = \lfloor \mu \rfloor \beta^{e-t}$, $y_2 = \lceil \mu \rceil \beta^{e-t}$, and hence $\mathsf{fl}(x) \in \{y_1, y_2\}$. Because x is rounded to the nearest floating point number, we have $|\mathsf{fl}(x) - x| \le |y_2 - y_1|/2 = \beta^{e-t}/2$, where the last equality follows from the definition of y_1, y_2. This implies that

$$\left| \frac{\mathsf{fl}(x) - x}{x} \right| \le \frac{\frac{\beta^{e-t}}{2}}{\mu \cdot \beta^{e-t}} \le \frac{1}{2}\beta^{1-t} = u,$$

establishing the claim of the theorem. □

[5]More precisely, one would need to write *a* nearest element.

The quantity $u = u(\mathbb{F}) := \frac{1}{2}\beta^{1-t}$ of Theorem 1.11 is called **unit roundoff** and features prominently in every round-off error analysis.

Example 1.12

Double	$u = 2^{-53} \approx 1.11 \times 10^{-16}$
Single	$u = 2^{-24} \approx 5.96 \times 10^{-8}$

In PYTHON u is computed by `sys.float_info.epsilon/2` and `numpy.finfo(numpy.float32).eps/2`, respectively.

The following variation of Theorem 1.11 is sometimes useful.

Theorem 1.13 *For every $x \in r(\mathbb{F})$ there exists $\delta \in \mathbb{R}$ such that*

$$\mathsf{fl}(x) = \frac{x}{1+\delta}, \quad |\delta| < u.$$

Proof. EFY. □

It is important to keep in mind that the quantity δ in Theorems 1.11 and 1.13 depends on x. In a round-off error analysis one does not operate with its explicit value but only with the property that $|\delta| < u$.

1.4 Elementary operations in $\mathbb{F}$ and round-off error

We now let '$\circ$' denote any of the four elementary binary operations:

$$\circ \in \{+, -, *, /\}.$$

The set $\mathbb{R}$ is closed under these operations, that is, with the exception of division by zero, $\circ$ maps two real numbers again to a real number. This closedness property does not hold for $\mathbb{F}$. To map the result back to $\mathbb{F}$, one needs to round. It would be reasonable to require a result of an elementary operation on a computer to be equal to what would have been obtained from *first computing the result exactly and then rounding it* afterwards. In practice, a slightly weaker requirements is imposed, which follows from combining the "first computing exactly then rounding" paradigm with Theorem 1.11.

Definition 1.14 (Standard model of rounding) *For every $x, y \in r(\mathbb{F})$ there exists $\delta \in \mathbb{R}$ such that*

$$\mathsf{fl}(x \circ y) = (x \circ y)(1+\delta), \quad |\delta| \le u, \quad \circ \in \{+, -, *, /\}. \tag{1.5}$$

In analogy to Theorem 1.13, an alternative to (1.5) is to require

$$\mathsf{fl}(x \circ y) = \frac{x \circ y}{1+\delta'}, \quad |\delta'| \le u, \quad \circ \in \{+, -, *, /\}. \tag{1.6}$$

It is important to remark, once more, that the quantities δ, δ' above depend on the inputs x, y and, in particular, they may also depend on the order of x, y. Although $\mathsf{fl}(x \circ y) = \mathsf{fl}(y \circ x)$ often holds in practice, commutativity is not and cannot be taken for granted when operating in floating point arithmetic. More importantly, associativity is frequently violated - even under the stronger "first computing exactly then rounding" paradigm.

Example 1.15 Let $\beta = 10$, $t = 2$. Then

$$\begin{aligned} & \mathsf{fl}(\mathsf{fl}(70 + 74) + 74) = \mathsf{fl}(140 + 74) = 210 \\ \neq \quad & \mathsf{fl}(70 + \mathsf{fl}(74 + 74)) = \mathsf{fl}(70 + 150) = 220 \end{aligned}$$

and

$$\begin{aligned} & \mathsf{fl}(\mathsf{fl}(110 - 99) - 10) = \mathsf{fl}(11 - 10) = 1 \\ \neq \quad & \mathsf{fl}(110 + \mathsf{fl}(-99 - 10)) = \mathsf{fl}(110 - 110) = 0. \end{aligned}$$

◇

The property (1.5) is also desirable for elementary functions $f \in \{\exp, \sin, \cos, \tan, \ldots\}$:

$$\mathsf{fl}(f(x)) = f(x)(1 + \delta), \quad |\delta| \le u. \tag{1.7}$$

The development and implementation of a numerical method that computes $f(x)$ in finite precision arithmetic such that the result satisfies (1.7) is by no means trivial. It is very difficult predict how many digits of $f(x)$ need to computed accurately such that the result obtained after rounding satisfies (1.7).[6]

1.5 Round-off error analysis

The standard model of rounding allows us to estimate the propagation of roundoff errors in a computation. As an important example let us consider the computation of the inner product of two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$, that is,

$$\mathbf{x}^{\mathsf{T}}\mathbf{y} = \sum_{k=1}^{n} x_k y_k \ .$$

For the partial sums $s_i = \sum_{k=1}^{i} x_k y_k$, $i = 1, 2, \ldots$, evaluated in floating point arithmetic $\mathbb{F}$ according to the standard model, one obtains

$$\begin{aligned} \widehat{s}_1 &= \mathsf{fl}(x_1 y_1) = x_1 y_1 (1 + \delta_1), \\ \widehat{s}_2 &= \mathsf{fl}(\widehat{s}_1 + x_2 y_2) = \big(\widehat{s}_1 + x_2 y_2 (1 + \delta_2)\big)(1 + \delta_3) \\ &= x_1 y_1 (1 + \delta_1)(1 + \delta_3) + x_2 y_2 (1 + \delta_2)(1 + \delta_3), \end{aligned}$$

[6] In the literature this effect is called *Table Maker's Dilemma*; see also `http://perso.ens-lyon.fr/jean-michel.muller/Intro-to-TMD.htm`.

where $|\delta_i| \le u$ holds for all δ_i. To simplify the notation, it is common practice to drop indices and let δ denote an arbitrary real number with $|\delta| \le u$. Using this convention, one gets

$$\begin{aligned} \widehat{s}_3 &= \mathsf{fl}(\widehat{s}_2 + x_3 y_3) = \big(\widehat{s}_2 + x_3 y_3(1+\delta)\big)(1+\delta) \\ &= x_1 y_1 (1+\delta)^3 + x_2 y_2 (1+\delta)^3 + x_3 y_3 (1+\delta)^2. \end{aligned}$$

By induction, we obtain

$$\widehat{s}_n = x_1 y_1 (1+\delta)^n + x_2 y_2 (1+\delta)^n + x_3 y_3 (1+\delta)^{n-1} + \cdots + x_n y_n (1+\delta)^2. \qquad (1.8)$$

The following lemma can be used to simplify this expression.

Lemma 1.16 *Let $|\delta_i| \le u(\mathbb{F})$ for $i = 1, \ldots, n$ and $n < 1/u(\mathbb{F})$. Then there exists $\theta_n \in \mathbb{R}$ such that*

$$\prod_{i=1}^{n} (1+\delta_i) = 1 + \theta_n, \quad \textit{where} \quad |\theta_n| \le \frac{nu(\mathbb{F})}{1 - nu(\mathbb{F})} =: \gamma_n(\mathbb{F}). \qquad (1.9)$$

Proof. The proof proceeds by induction. For $n = 1$, the statement trivially holds. Induction hypothesis: The statement holds for $n-1$ with $n \ge 2$.

Induction step: Using the induction hypothesis, one obtains

$$\prod_{i=1}^{n} (1+\delta_i) = (1+\delta_n)(1+\theta_{n-1}) =: 1 + \theta_n \quad , \quad \theta_n := \delta_n + (1+\delta_n)\theta_{n-1},$$

$$|\theta_n| \le u + (1+u)\frac{(n-1)u}{1-(n-1)u} = \frac{nu}{1-(n-1)u} \le \gamma_n \ .$$

□

Applying Lemma 1.16 to (1.8) gives

$$\widehat{s}_n = x_1 y_1 (1+\theta_n) + x_2 y_2 (1+\theta_n') + x_3 y_3 (1+\theta_{n-1}) + \cdots + x_n y_n (1+\theta_2), \qquad (1.10)$$

where $|\theta_j| \le \gamma_j$ and $|\theta_n'| \le \gamma_n$. Using monotonicity, $0 < \gamma_j \le \gamma_n$, one therefore obtains

$$\widehat{s}_n = (\mathbf{x} + \triangle\mathbf{x})^{\mathsf{T}} \mathbf{y} = \mathbf{x}^{\mathsf{T}} (\mathbf{y} + \triangle\mathbf{y}), \quad |\triangle\mathbf{x}| \le \gamma_n |\mathbf{x}|, \quad |\triangle\mathbf{y}| \le \gamma_n |\mathbf{y}|, \qquad (1.11)$$

where $|\mathbf{x}|$ is the vector with elements $|x_i|$ and inequalities between vectors and matrices are understood componentwise. Let us remark that the bounds in (1.11) are – in contrast to (1.10) – independent of the order of summation.

The result (1.11) is an example for what is called a **backward error**. It states that the computed result (inner product) is the exact inner product of slightly perturbed input data ($\mathbf{x}$ and $\mathbf{y}$). Since the inputs are usually perturbed by roundoff error anyway (for example when storing the data as floating point numbers), a small backward error is a very desirable property. It implies that the algorithm attains an accuracy (nearly) at the level of the accuracy of the input data. The actual error in the computed result is called **forward error**. From (1.11) one obtains the following bound on the forward error:

$$\left|\mathbf{x}^{\mathsf{T}}\mathbf{y} - \widehat{s}_n\right| \le \gamma_n \sum_{i=1}^{n} |x_i y_i| = \gamma_n |\mathbf{x}|^{\mathsf{T}}|\mathbf{y}|. \tag{1.12}$$

If $|\mathbf{x}^{\mathsf{T}}\mathbf{y}| \approx |\mathbf{x}|^{\mathsf{T}}|\mathbf{y}|$ then the relative error $\left|\mathbf{x}^{\mathsf{T}}\mathbf{y} - \mathsf{fl}(\mathbf{x}^{\mathsf{T}}\mathbf{y})\right|/|\mathbf{x}^{\mathsf{T}}\mathbf{y}|$ is small. If, on the other hand, $|\mathbf{x}^{\mathsf{T}}\mathbf{y}| \ll |\mathbf{x}|^{\mathsf{T}}|\mathbf{y}|$ then one cannot expect a small relative error.

1.6 Cancellation

Numerical cancellation happens when two numbers that are nearly equal *and* already affected by roundoff error are subtracted from each other. Cancellation is the number one reason to look for when errors get massively amplified in the course of a computation.

Example 1.17 Consider

$$f(x) = \frac{1-\cos(x)}{x^2}\,, \quad x = 1.2 \times 10^{-5}.$$

Rounding $\cos(x)$ to 10 decimal digits gives

$$\widehat{c} = 0.9999\,9999\,99\,,$$

and

$$1 - \widehat{c} = 0.0000\,0000\,01\,.$$

Therefore $(1-\widehat{c})/x^2 = 10^{-10}/1.44 \times 10^{-10} = 0.6944\ldots$. However, since we know that $0 \le f(x) < \frac{1}{2}$ for $x \neq 0$ it is obvious that the computed result is completely wrong. ◇

To see the general picture, let us consider $a, b \in \mathbb{R}$ and

$$\widehat{a} = \mathsf{fl}(a) = a(1+\delta_a), \quad \widehat{b} = \mathsf{fl}(b) = b(1+\delta_b)$$

with $|\delta_a| \le u$, $|\delta_b| \le u$. Then $x = a - b$ and $\widehat{x} = \widehat{a} - \widehat{b}$ satisfy

$$\left|\frac{x-\widehat{x}}{x}\right| = \left|\frac{-a\delta_a + b\delta_b}{a-b}\right| \le u\,\frac{|a|+|b|}{|a-b|} \tag{1.13}$$

The relative error in the computation $x = a - b$ is large when

$$|a-b| \ll |a| + |b|.$$

Let us emphasize that it is crucial in the discussion above that a and b are already affected by roundoff error. The subtraction itself is carried out *without roundoff error* for $a \approx b$; it just amplifies the existing errors.

Roundoff errors can also cancel each other, that is, a very inaccurate intermediate result does not necessarily lead to very inaccurate final result. This effect is demonstrated by the following example.

x	Alg. 1	Alg. 2
10^{-3}	*1.0005*236	*1.0005002*
10^{-4}	*1.0001*659	*1.0000499*
10^{-5}	*1.00*13580	*1.0000050*
10^{-6}	0.9536743	*1.0000005*
10^{-7}	*1.*1920929	*1.0000001*

Table 1.2: Results of Algorithms 1 and 2 in single precision. Correctly computed digts are italic.

Example 1.18 (Computation of $(e^x - 1)/x$ for $x \to 0+$) We aim at computing

$$f(x) = (e^x - 1)/x = \sum_{i=0}^{\infty} \frac{x^i}{(i+1)!} \tag{1.14}$$

in IEEE single precision by two different methods:

Python
```
# Algorithm 1
import numpy as np
if x == 0:
    f = 1
else:
    f = ( np.exp(x) - 1 ) / x;
```

Python
```
# Algorithm 2
import numpy as np
y = np.exp(x)
if y == 1:
    f = 1
else:
    f = (y - 1) / np.log(y);
```

The obtained results are shown in Table 1.2.[7] To understand why Algorithm 1 yields much worse accuracy, we consider $x = 9.0 \times 10^{-8}$ and assume that the implementations of the elementary functions $\exp(\cdot)$ and $\log(\cdot)$ satisfy the standard model (1.7). The first 9 decimal digits of the exact result are

$$\frac{e^x - 1}{\log e^x} = 1.00000005\,.$$

Algorithm 1 yields

$$\mathsf{fl}\Big(\frac{\mathsf{fl}(\mathsf{fl}(e^x) - 1)}{x}\Big) = \mathsf{fl}\Big(\frac{1.19209290 \times 10^{-7}}{9.\overline{0} \qquad\quad \times 10^{-8}}\Big) = 1.32454766\,;$$

In contrast, Algorithm 2 yields

$$\mathsf{fl}\Big(\frac{\mathsf{fl}(\widehat{y} - 1)}{\mathsf{fl}(\log \widehat{y})}\Big) = \mathsf{fl}\Big(\frac{1.19209290 \times 10^{-7}}{1.19209282 \times 10^{-7}}\Big) = 1.00000006\,.$$

[7] It can be difficult to reproduce these results, especially when working in compiled languages. Some processors work internally (registers) with higher-precision 80-bit arithmetic; so the results may depend on whether the registers are flushed, which in turn depends on the compiler, etc. The behavior might also change when using a debugger or displaying intermediate results, leading to the notorious heisenbugs.

One observes that Algorithm 2 computes, because of cancellation, a very inaccurate result for the numerator $e^x - 1 = 9.00000041 \times 10^{-8}$ and the denominator $\log e^x = 9 \times 10^{-8}$; but most of the roundoff error vanishes during the division!

The described phenomenon can be explained by a roundoff error analysis of Algorithm 2. We have $\widehat{y} = e^x(1+\delta)$, $|\delta| \le u$. If $\widehat{y} = 1$, it follows that

$$e^x(1+\delta) = 1 \iff x = -\log(1+\delta) = -\delta + \delta^2/2 - \delta^3/3 + \dots\,,$$

and hence

$$\widehat{f} = \mathsf{fl}\big(1 + x/2 + x^2/6 + \dots\big|_{x=-\delta+O(\delta^2)}\big) = 1\,. \tag{1.15}$$

If $\widehat{y} \ne 1$, it follows that

$$\widehat{f} = \mathsf{fl}\big((\widehat{y}-1)/\log\widehat{y}\big) = \frac{(\widehat{y}-1)(1+\delta_1)}{\log\widehat{y}(1+\delta_2)}\,(1+\delta_3), \quad |\delta_i| \le u\,. \tag{1.16}$$

Defining $v := \widehat{y} - 1$, we obtain

$$\begin{aligned} g(\widehat{y}) :&= \frac{\widehat{y}-1}{\log\widehat{y}} = \frac{v}{\log(1+v)} = \frac{v}{v - v^2/2 + v^3/3 - \dots} \\ &= \frac{1}{1 - v/2 + v^2/3 - \dots} = 1 + \frac{v}{2} + O(v^2)\,. \end{aligned}$$

For small x, we have $y \approx 1$ and

$$g(\widehat{y}) - g(y) \approx \frac{\widehat{y}-y}{2} \approx \frac{e^x\delta}{2} \approx \frac{\delta}{2} \approx g(y)\,\frac{\delta}{2}.$$

By (1.16),

$$\Big|\frac{\widehat{f}-f}{f}\Big| \le 3.5\,u\,. \tag{1.17}$$

While $\widehat{y} - 1$ and $\log\widehat{y}$ are very inaccurate, the quantity $(\widehat{y}-1)/\log\widehat{y}$ is a very accurate approximation of $(y-1)/\log y$ at $y = 1$, because $g(y) := (y-1)/\log y$ varies “slowly” close to 1. $\diamond$