Chapter 1

Computer arithmetic

In this chapter, we give an elementary overview of how (and what type of) numbers are represented, stored and manipulated in a computer. This will provide some insight as to why some floating point computations produce grossly incorrect results. This topic is covered much more extensively in e.g. [Hi96], [Ov01], and [Wi63].

1.1 Positional systems

Our everyday decimal number system is a *positional system* in base 10. Since computer arithmetic is often built on positional systems in other bases (e.g. 2 or 16)[1], we will begin this section by recalling how real numbers are represented in a positional system with an arbitrary integer base $\beta \geq 2$. Setting aside practical restrictions, such as the finite storage capabilities of a computer, any real number can be expressed as an infinite string

$$(-1)^\sigma(b_n b_{n-1} \dots b_0 . b_{-1} b_{-2} \dots)_\beta, \tag{1.1}$$

where $b_n, b_{n-1}, \dots$ are integers in the range $[0, \beta - 1]$, and $\sigma \in \{0, 1\}$ provides the sign of the number. The real number corresponding to (1.1) is

$$\begin{aligned} x &= (-1)^\sigma \sum_{i=-\infty}^{n} b_i \beta^i \\ &= (-1)^\sigma (b_n \beta^n + b_{n-1}\beta^{n-1} + \dots + b_0 + b_{-1}\beta^{-1} + b_{-2}\beta^{-2} + \dots). \end{aligned}$$

If the number ends in an infinite number of consecutive 0:s we omit them in the expression (1.1). Thus we write $(12.25)_{10}$ instead of $(12.25000\dots)_{10}$. Also, we omit any 0:s preceding the integer part $(-1)^\sigma(b_n b_{n-1} \dots b_0)_\beta$. Thus we write $(12.25)_{10}$

[1]Of course, some exceptions do exist: most calculators still use base 10; the Russian computer *Setun* used base 3, whereas the American *Maniac II* used base $65536 = 16^4$!

instead of $(0012.25)_{10}$, and $(0.0025)_{10}$ instead of $(000.0025)_{10}$. Allowing for either leading or trailing extra 0:s is called *padding*, and is not common practice since it leads to redundancies in the representation.

Even without padding, the positional system is slightly flawed. No matter what base we choose, there are still real numbers that do not have a unique representation. For example, the decimal number $(12.2549999\dots)_{10}$ is equal to $(12.255)_{10}$, and the binary number $(100.01101111\dots)_2$ is equal to $(100.0111)_2$. This redundancy, however, can be avoided if we add the requirement that $0 \le b_i \le \beta - 2$ for infinitely many i.

Exercise 1.1.1 *Prove that any real number $x \neq 0$ has a unique representation (1.1) in a positional system (allowing no padding) with integer base $\beta \ge 2$ under the two conditions (a) $0 \le b_i \le \beta - 1$ for all i, and (b) $0 \le b_i \le \beta - 2$ for infinitely many i.*

Exercise 1.1.2 *What is the correct way to represent zero in a positional system allowing no padding?*

1.2 Floating point numbers

When expressing a real number on the form (1.1), the placement of the decimal[2] point is crucial. The *floating point* number system provides a more convenient way to represent real numbers. A floating point number is a real number on the form

$$x = (-1)^\sigma m \times \beta^e, \tag{1.2}$$

where $(-1)^\sigma$ is the sign of x, m is called the *mantissa*[3], and e is called the *exponent* of x. Writing numbers in floating point notation frees us from the burden of keeping track of the decimal point: it always follows the first digit of the mantissa. It is customary to write the mantissa as

$$m = (b_0.b_1b_2\dots)_\beta$$

where, compared to the previous section, the indexing of the b_i has opposite sign. As this is the standard notation, we will adopt this practice in what follows. Thus we may define the set of floating point numbers in base β as:

$$\mathbb{F}_\beta = \{(-1)^\sigma m \times \beta^e \colon m = (b_0.b_1b_2\dots)_\beta\},$$

where, as before, we request that β is an integer no less than 2, and that $0 \le b_i \le \beta - 1$ for all i, and $0 \le b_i \le \beta - 2$ for infinitely many i. The exponent e may be any integer.

[2]Being picky, the expression "*base* point" or "*radix* point" is more appropriate, unless $\beta = 10$.

[3]The mantissa is sometimes referred to as the *significand* or, rather incorrectly, as the *fractional part* of the floating point number.

Expressing real numbers in floating point form introduces a new type of redundancy. For example, the base-10 number 123 can be expressed as $(1.23)_{10} \times 10^2$, $(0.123)_{10} \times 10^3$, $(0.0123)_{10} \times 10^4$, and so on. In order to have unique representations for non-zero real numbers, we demand that the leading digit b_0 be non-zero, except for the special case $x = 0$. Floating point numbers satisfying this additional requirement are said to be *normal* or *normalized*[4].

Exercise 1.2.1 *Show that a non-zero floating point number is normal iff its associated exponent e is chosen minimal.*

So far, we have simply toyed with different representations of the real numbers $\mathbb{R}$. As this set is uncountably infinite (see Theorem A.3.6), whereas a machine can only store a finite amount of information, more drastic means are called for: we must introduce a new, much smaller, set of numbers designed to fit into a computer, and which at the same time approximate the real numbers in some well-defined sense.

As a first step toward this goal, we restrict the number of digits representing the mantissa. This yields the set

$$\mathbb{F}_{\beta,p} = \{x \in \mathbb{F}_\beta : m = (b_0.b_1 b_2 \dots b_{p-1})_\beta\}.$$

The number p is called the *precision* of the floating point system. It is a nice exercise to show that, although $\mathbb{F}_{\beta,p}$ is a much smaller set than $\mathbb{F}_\beta$, it is countably infinite (see Theorem A.2.1). This means that even $\mathbb{F}_{\beta,p}$ is too large for our needs. Note, however, that the restriction $0 \le b_i \le \beta - 2$ for infinitely many i becomes void in finite precision.

A *finite* set of floating point numbers can be formed by imposing a fixed precision, as well as bounds on the admissible exponents. Such a set is specified by four integers: the base β, the precision p, and the minimal and maximal exponents $\check{e}$ and $\hat{e}$, respectively. Given these quantities we can define parameterized sets of *computer representable* floating point numbers:

$$\mathbb{F}_{\beta,p}^{\check{e},\hat{e}} = \{x \in \mathbb{F}_{\beta,p} : \check{e} \le e \le \hat{e}\}.$$

Exercise 1.2.2 *Show that $\mathbb{F}_{\beta,p}^{\check{e},\hat{e}}$ is finite, whereas $\mathbb{F}_{\beta,p}$ is countably infinite, and $\mathbb{F}_\beta$ is uncountably infinite, with*

$$\mathbb{F}_{\beta,p}^{\check{e},\hat{e}} \subset \mathbb{F}_{\beta,p} \subset \mathbb{F}_\beta.$$

Exercise 1.2.3 *How many normal numbers belong to the set $\mathbb{F}_{\beta,p}^{\check{e},\hat{e}}$?*

[4]This should not be confused with the number-theoretic notion of a normal number. There a number is normal to base β if every sequence of n consecutive digits in its β-expansion appears with limiting probability β^{-n}.

Using a base other than 10 forces us to have to rethink which numbers have a finite representation. This can cause some confusion to the novice programmer.

Example 1.2.4 *With $\beta = 2$ and $p < \infty$, the number $1/10$ is not exactly representable in $\mathbb{F}_{\beta,p}$. This can be seen by noting that*

$$\sum_{k=1}^{\infty} \left(2^{-4k} + 2^{-(4k+1)}\right) = \frac{3}{2}\left(\frac{1}{1-2^{-4}} - 1\right) = \frac{1}{10}.$$

Interpreting the first sum as a binary representation, we have

$$1/10 = (0.00011001100110011\ldots)_2 = (1.1001100110011\ldots)_2 \times 2^{-4}.$$

Since this is a non-terminating binary number, it has no exact representation in $\mathbb{F}_{2,p}$, regardless of the choice of precision p.

This example may come as a surprise to all who use the popular step-sizes 0.1 or 0.001 in, say, numerical integration methods. Most computers use $\beta = 2$ in their internal floating point representation, which means that on these computers $1000 \times 0.001 \neq 1$. This simple fact can have devastating consequences for e.g. very long integration procedures. More suitable step-sizes would be $2^{-3} = 0.125$ and $2^{-10} = 0.0009765625$, which are exactly representable in base two. Example 1.4.2 further illustrates how sensitive numerical computations can be to these small conversion errors.

1.2.1 Subnormal numbers

As mentioned earlier, without the normalizing restriction $b_0 \neq 0$, a non-zero real number may have several representations in the set $\mathbb{F}_{\beta,p}^{\check{e},\hat{e}}$. (Note, however, that most real numbers no longer have *any* representation in this finite set.) We already remarked that these redundancies may be avoided by normalization, i.e., by demanding that all non-zero floating point numbers have a non-zero leading digit. To illustrate the concept of normalized numbers, we will study a small toy system of floating point numbers, illustrated in Figure 1.1.

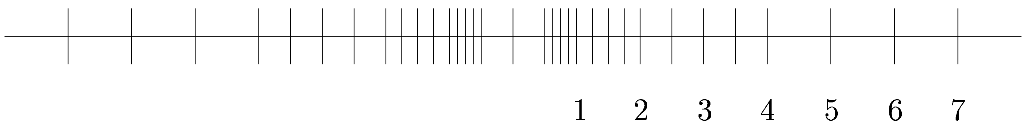

Figure 1.1: The normal numbers of $\mathbb{F}_{2,3}^{-1,2}$.

It is clear that the floating point numbers are not uniformly distributed: the positive numbers are given by $\{0.5, 0.625, 0.75, 0.875, 1, 1.25, 1.5, 1.75, 2, 2.5, 3, 3.5, 4, 5, 6, 7\}$. Thus the intermediate distances between consecutive numbers range within the set

$\{0.125, 0.25, 0.5, 1\}$. In Section 1.3, we will explain why this type of non-uniform spacing is a good idea. We will also describe how to deal with real numbers having modulus greater than the largest positive normal number N^n_{max}, which in our toy system is $(1.11)_2 \times 2^2 = 7$.

Note that the smallest positive normal number in $\mathbb{F}^{-1,2}_{2,3}$ is $N^n_{min} = (1.00)_2 \times 2^{-1} = 0.5$, which leaves an undesired gap centered around the origin. Not only does this lead to a huge loss of accuracy when approximating numbers of small magnitude, it also leads to the violation of some of our most valued mathematical laws, see Exercise 1.4.5. A way to work around these problems is to allow for some numbers that are not normal.

A non-zero floating point number in $\mathbb{F}^{\check{e},\hat{e}}_{\beta,p}$ with $b_0 = 0$ and $e = \check{e}$, it is said to be *subnormal* (or denormalized). Subnormal numbers allow for a *gradual underflow* to zero (compare Figures 1.2 and 1.1). Extending the set of admissible floating point numbers to include the subnormal numbers still preserves uniqueness of representation, although the use of these additional numbers comes with some penalties, as we shall shortly see. For our toy system at hand, the positive subnormal numbers are $\{0.125, 0.25, 0.375\}$.

Figure 1.2 illustrates the normal and subnormal numbers of $\mathbb{F}^{-1,2}_{2,3}$:

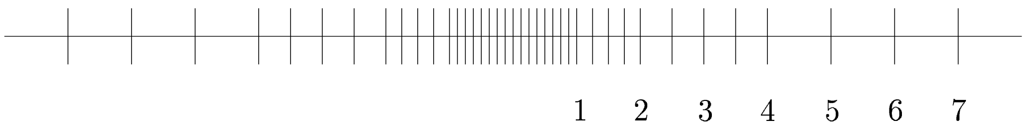

Figure 1.2: The normal and subnormal numbers of $\mathbb{F}^{-1,2}_{2,3}$.

The difference between these two sets is striking: the smallest positive normal number N^n_{min} is $(1.00)_2 \times 2^{-1} = 0.5$, whereas the smallest positive subnormal number N^s_{min} is $(0.01)_2 \times 2^{-1} = 0.125$. The largest subnormal number N^s_{max} is $(0.11)_2 \times 2^{-1} = 0.375$. Geometrically, introducing subnormal numbers corresponds to filling the gap centered around the origin with evenly spaced numbers. The spacing should be the same as between the two smallest positive normal numbers.

From now on, when we refer to the set $\mathbb{F}^{\check{e},\hat{e}}_{\beta,p}$, we mean the set of normal *and* subnormal numbers. We will use $\mathbb{F}$ to denote any set of type $\mathbb{F}^{\check{e},\hat{e}}_{\beta,p}$ or $\mathbb{F}_{\beta,p}$; when needed, the exact parameters of the set in question will be provided. A real number x with $N^n_{min} \leq |x| \leq N^n_{max}$ is said to be in the *normalized range* of the associated floating point system.

Exercise 1.2.5 *Prove that the non-zero normal and subnormal numbers of* $\mathbb{F}^{\check{e},\hat{e}}_{\beta,p}$ *have unique representations.*

Exercise 1.2.6 *How many positive subnormal numbers are there in the set* $\mathbb{F}^{\check{e},\hat{e}}_{\beta,p}$ *?*

Exercise 1.2.7 *Derive formulas for N^s_{min}, N^s_{max}, N^n_{min}, and N^n_{max} for a general set $\mathbb{F}^{\check{e},\hat{e}}_{\beta,p}$.*

We conclude this section by remarking that, although the floating point numbers are not uniformly spaced, for $\check{e} \leq m, n < \hat{e}$, the sets $\mathbb{F}^{\check{e},\hat{e}}_{\beta,p} \cap [\beta^m, \beta^{m+1}]$ and $\mathbb{F}^{\check{e},\hat{e}}_{\beta,p} \cap [\beta^n, \beta^{n+1}]$ have the same cardinality. This is apparent in Figures 1.2 and 1.1. We also note that any floating point system $\mathbb{F}$ is symmetric with respect to the origin: $x \in \mathbb{F} \Leftrightarrow -x \in \mathbb{F}$.

Exercise 1.2.8 *Compute the number of elements of the set $\mathbb{F}^{\check{e},\hat{e}}_{\beta,p} \cap [\beta^n, \beta^{n+1})$, provided that $\check{e} \leq n < \hat{e}$.*

1.3 Rounding

We have now reached the stage where we we have succeeded to condense the uncountable set of real numbers $\mathbb{R}$ into a finite set of floating point numbers $\mathbb{F}$. Almost all commercial computers use a set like $\mathbb{F}$, with some minor additions, to approximate the real numbers. In order to make computing feasible over $\mathbb{F}$, we must find a means to associate any real number $x \in \mathbb{R}$ to a member y of $\mathbb{F}$. Such an association is called *rounding*, and defines a map from $\mathbb{R}$ onto $\mathbb{F}$. Obviously, we cannot make the map invertible, but we would like to make it as close as possible to a homeomorphism.

Before defining such a mapping, we will extend both the domain and range into the sets $\mathbb{R}^* = \mathbb{R} \cup \{-\infty, +\infty\}$ and $\mathbb{F}^* = \mathbb{F} \cup \{-\infty, +\infty\}$, respectively. This provides an elegant means for representing real numbers that are too large in absolute value to fit into $\mathbb{F}$. In the actual implementation, the symbols $\{-\infty, +\infty\}$ are specially coded, and do not have a valid representation such as (1.2).

Following the excellent treatise on computer arithmetic [KM81], a rounding operation $\bigcirc\colon \mathbb{R}^* \to \mathbb{F}^*$ should have the following properties:

$$\text{(R1)} \qquad x \in \mathbb{F}^* \Rightarrow \bigcirc(x) = x,$$

$$\text{(R2)} \qquad x, y \in \mathbb{R}^* \text{ and } x \leq y \Rightarrow \bigcirc(x) \leq \bigcirc(y).$$

Property (R1) simply states that all members of $\mathbb{F}^*$ are fixed under $\bigcirc$. Clearly an already representable number does not require any further rounding. Property (R2) states that the rounding is monotone. Indeed, it would be very difficult to interpret the meaning of any numerical procedure without this property. Combining (R1) and (R2), one can easily prove that the rounding $\bigcirc$ is of *maximum quality*, i.e., the interior of the interval spanned by x and $\bigcirc(x)$ contains no points of $\mathbb{F}^*$.

Lemma 1.3.1 *Let $x \in \mathbb{R}^*$. If $\bigcirc\colon \mathbb{R}^* \to \mathbb{F}^*$ satisfies both (R1) and (R2), then the interval spanned by x and $\bigcirc(x)$ contains no points of $\mathbb{F}^*$ in its interior.*

Proof: The claim is trivially true if $x = \bigcirc(x)$ (since then the interior is empty), so assume that $x \neq \bigcirc(x)$. Without loss of generality we may assume that $x < \bigcirc(x)$. Now suppose that the claim is false, i.e., there exists an element $y \in \mathbb{F}^*$ with $x < y < \bigcirc(x)$. Since, by (R1), we have $\bigcirc(y) = y$, we must, by (R2), have $\bigcirc(x) \leq y$. This gives the desired contradiction. $\square$

We will now describe four rounding modes that are available on most commercial computers

1.3.1 Round to zero

The simplest rounding operation to implement is *round toward zero*, often referred to as *truncation*, which we denote by $\square_z$. We formally define $\square_z\colon \mathbb{R}^* \to \mathbb{F}^*$ by

$$\square_z(x) = \mathrm{sign}(x)\max\{y \in \mathbb{F}^* : y \leq |x|\}, \tag{1.3}$$

where $\mathrm{sign}(x)$ is the sign of x. The action of $\square_z$ is illustrated in Figure 1.3. To see how easy this rounding mode is to implement, consider a real number $x = (-1)^\sigma(b_0.b_1b_2\dots)_\beta \times \beta^e$ in $\mathbb{F}_\beta$ to be rounded into $\mathbb{F}_{\beta,p}$. If x satisfies $|x| \leq N^n_{max}$, this is achieved by simply discarding the mantissa digits beyond position $p-1$ (hence the nickname truncation): $\square_z(x) = (-1)^\sigma(b_0.b_1b_2\dots b_{p-1})_\beta \times \beta^e$. Otherwise, x is rounded to $(-1)^\sigma N^n_{max}$.

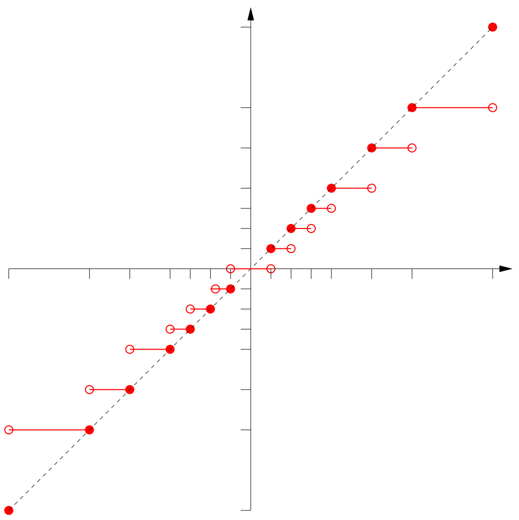

Figure 1.3: Round to zero $\square_z$.

Round toward zero is an *odd*[5] function:

[5]A function f is said to be *odd* if $f(-x) = -f(x)$, and *even* if $f(-x) = f(x)$. Most functions are neither.

(R3) $x \in \mathbb{R}^* \Rightarrow \bigcirc(-x) = - \bigcirc(x).$

Rule (R3) is also satisfied by the most common rounding mode: *round to nearest*, which we shall return to later.

1.3.2 Directed rounding

There are two very useful rounding modes that are said to be *directed*. By this we mean that they satisfy (in addition to (R1) and (R2)) one of the following rules:

(R4) (a) $x \in \mathbb{R}^* \Rightarrow \bigcirc(x) \leq x$ or (b) $x \in \mathbb{R}^* \Rightarrow \bigcirc(x) \geq x.$

The rounding mode satisfying (R4a) is called *round toward minus infinity* (or simply *round down*). The rounding mode satisfying (R4b) is called *round toward plus infinity* (or simply *round up*). These rounding modes are denoted $\bigtriangledown(x)$ and $\bigtriangleup(x)$, respectively, and are formally defined by

$$\bigtriangledown(x) = \max\{y \in \mathbb{F}^* : y \leq x\} \quad \text{and} \quad \bigtriangleup(x) = \min\{y \in \mathbb{F}^* : y \geq x\}. \qquad (1.4)$$

The number $\bigtriangledown(x)$, called x *rounded down*, is the largest floating point number less than or equal to x, whereas $\bigtriangleup(x)$, called x *rounded up*, is the smallest floating point number greater than or equal to x. Note that, if $x \in \mathbb{F}^*$, then $\bigtriangledown(x) = x = \bigtriangleup(x)$, whereas if $x \notin \mathbb{F}^*$, we have the enclosure $\bigtriangledown(x) < x < \bigtriangleup(x)$, which is of maximal quality. This means that the interval $[\bigtriangledown(x), \bigtriangleup(x)]$ contains no points of $\mathbb{F}^*$ in its interior (apply Lemma 1.3.1 twice). Also note the anti-symmetry relations:

$$\bigtriangleup(-x) = - \bigtriangledown(x) \quad \text{and} \quad \bigtriangledown(-x) = - \bigtriangleup(x). \qquad (1.5)$$

Thus either rounding $\bigtriangledown$ or $\bigtriangleup$ can be completely defined in terms of the other.

Exercise 1.3.2 *Using only (1.4), prove that the rounding modes $\bigtriangledown$ and $\bigtriangleup$ satisfy (R1) and (R2). Also show that $\bigtriangledown$ satisfies (R4a), and that $\bigtriangleup$ satisfies (R4b).*

Exercise 1.3.3 *Show that, in terms of the directed rounding mode $\bigtriangledown$, we have*

$$\square_z(x) = \mathrm{sign}(x) \bigtriangledown (|x|).$$

The relations (1.4) completely define the directed roundings $\bigtriangledown$ and $\bigtriangleup$ as maps from $\mathbb{R}^*$ to $\mathbb{F}^*$, see Figure 1.4.

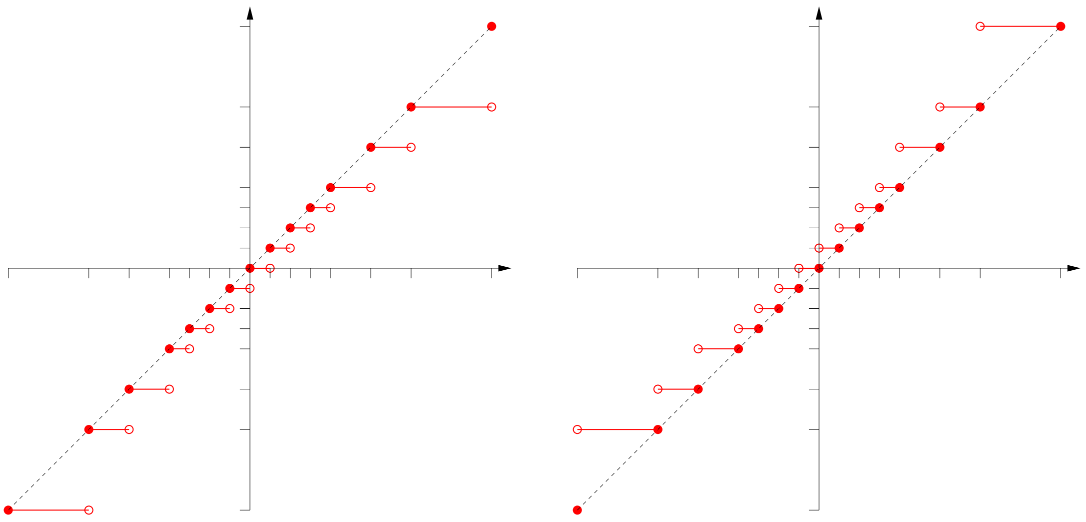

Figure 1.4: Directed roundings: (a) round down $\triangledown$; (b) round up $\triangle$.

1.3.3 Round to nearest (even)

Note that all previously defined rounding modes map the interior any interval spanned by two consecutive floating point numbers onto a single point in $\mathbb{F}^*$. This means that the error made when rounding a single real number x could be as large as the length of the interval $[\triangledown(x), \triangle(x)]$ enclosing it. A more accurate family of rounding modes is called *round to nearest*.

For an element of $x \in \mathbb{R}^*$, we can construct an enclosure of x in $\mathbb{F}^*$:

$$\triangledown(x) \leq x \leq \triangle(x).$$

If also $|x| \leq N^n_{max}$, we let $\mu = \frac{1}{2}(\triangle(x) + \triangledown(x))$ denote the midpoint of this interval. If $|x| > N^n_{max}$, we let $\mu = \mathrm{sign}(x)N^n_{max}$. Rounding to nearest returns $\triangledown(x)$ if $x < \mu$, and $\triangle(x)$ if $x > \mu$. In the rare case $x = \mu$, there is a tie. The different variants of rounding to nearest are distinguished according to how they resolve this tie.

One easy way to break the tie, is to simply round up for positive ties, and down for negative ones. This rounding mode is called *round to nearest*, and is defined by

$$\begin{aligned} x > 0 \Rightarrow \Box_n(x) &= \begin{cases} \triangledown(x), & \text{if } x \in [\triangledown(x), \mu), \\ \triangle(x), & \text{if } x \in [\mu, \triangle(x)], \end{cases} \\ x < 0 \Rightarrow \Box_n(x) &= -\Box_n(-x). \end{aligned} \tag{1.6}$$

Although easy to describe, this rounding mode has the slight disadvantage of being *biased*: the rounding errors are not evenly distributed around zero. Indeed, if x is positive, then $\Box_n$ has a higher probability of rounding x downward, and vice-versa, see Figure 1.5a.

An unbiased way to break the tie gives rise to a rounding mode called *round to nearest even*, which we simply denote by $\Box$. This is the default rounding mode on almost all commercial computers. In order to define this rounding operation, we will assume that the mantissas of $\bigtriangledown(x)$ and $\bigtriangleup(x)$ are given by

$$(a_0.a_1a_2\dots a_{p-1})_\beta \quad \text{and} \quad (b_0.b_1b_2\dots b_{p-1})_\beta,$$

respectively. Note that, by Lemma 1.3.1, if $x \notin \mathbb{F}^*$ then *exactly* one of the integers a_{p-1} and b_{p-1} is even. We define the *round to nearest even* scheme by

$$x > 0 \Rightarrow \Box(x) = \begin{cases} \bigtriangledown(x), & \text{if } x \in [\bigtriangledown(x), \mu), \text{ or if } x = \mu \text{ and } a_{p-1} \text{ is even,} \\ \bigtriangleup(x), & \text{if } x \in (\mu, \bigtriangleup(x)], \text{ or if } x = \mu \text{ and } b_{p-1} \text{ is even,} \end{cases} \tag{1.7}$$
$$x < 0 \Rightarrow \Box(x) = -\Box(-x).$$

Note that this definition evens out the probability of rounding upward or downward; the rounding is *unbiased*, see Figure 1.5b.

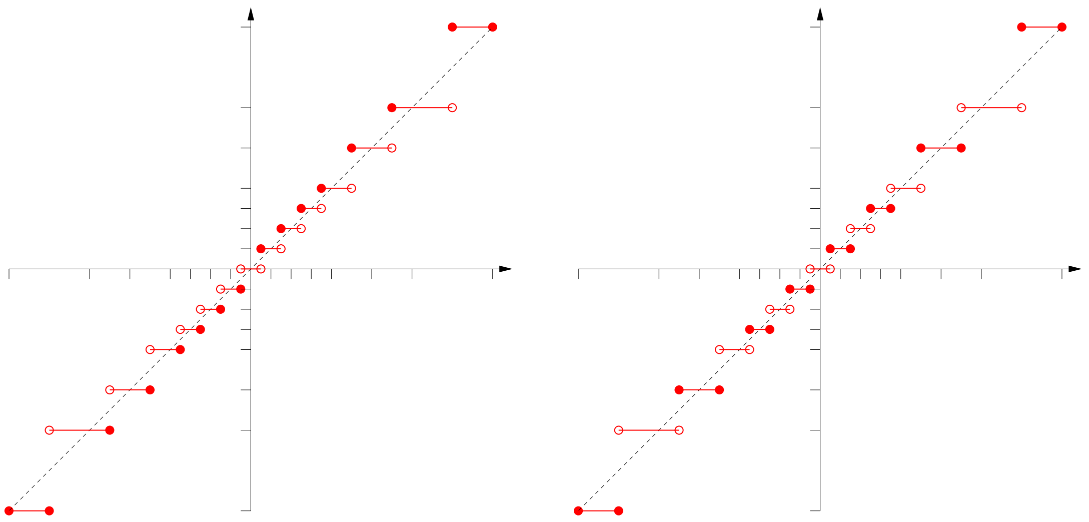

Figure 1.5: Round to nearest: (a) biased $\Box_n$; (b) unbiased $\Box$.

When rounding a real number x of very large magnitude ($|x| > N^n_{max}$) it should be pointed out that, despite the name *round to nearest*, x is actually rounded to $\text{sign}(x)\infty$, while $\text{sign}(x)N^n_{max}$ actually is the nearest element in $\mathbb{F}^*$.

As a final remark, it is clear that round to nearest *odd* can be defined in a completely analogous manner. One may then wonder whether the choice between rounding to nearest even or to nearest odd is relevant. The answer is, somewhat surprisingly, Yes! This is demonstrated in the following example.

Example 1.3.4 *Consider the following scenario: let $\beta = 10$, $x = 0.45$, and suppose that we want to round x to two digits, after which we continue to round the result to* *one digit. Using round to nearest even produces* $0.45 \to 0.4 \to 0 = \tilde{x}_e$*, whereas round to nearest odd produces* $0.45 \to 0.5 \to 1 = \tilde{x}_o$*, which is not the nearest single-digit number seeing that* $|x - \tilde{x}_e| = 0.45 < 0.55 = |x - \tilde{x}_o|$.

Now, suppose that $\beta = 4$, $x = (0.22)_4$*, and suppose once again that we want to round* x *to two digits, after which we continue to round the result to one digit. Using round to nearest even produces* $(0.22)_4 \to (0.2)_4 \to (0)_4 = \tilde{x}_e$*, whereas round to nearest odd produces* $(0.22)_4 \to (0.3)_4 \to (1)_4 = \tilde{x}_o$*, which now is the nearest single-digit number seeing that* $|x - \tilde{x}_e| = (0.22)_4 = 5/8 > 3/8 = (0.21)_4 = |x - \tilde{x}_o|$.

This example illustrates the fact that, when using an even base β, with $\beta/2$ *odd*, the best choice is round to nearest *even*. For an even base β, with $\beta/2$ *even*, however, the best choice is round to nearest *odd*. If the base itself β is odd, we can never have a tie (at least not in finite precision), so the choice of rounding never arises.

In particular, this means that for the popular choices $\beta = 10$ or 2, we should use round to nearest *even*, whereas for $\beta = 16$, round to nearest *odd* is the superior choice.

1.3.4 Rounding errors

In the normalized range, the error produced when rounding a real number to a floating point system can be bounded in terms of the base β and precision p. We have the following bounds on the relative and absolute rounding errors:

Theorem 1.3.5 *If* x *is a real number in the normalized range of* $\mathbb{F} = \mathbb{F}_{\beta,p}$*, then the relative error caused by rounding is bounded by* $\varepsilon_M = \beta^{-(p-1)}$*:*

$$\left| \frac{x - \bigcirc(x)}{x} \right| < \varepsilon_M.$$

Equivalently, the corresponding absolute error is bounded by $|x|\varepsilon_M$*:*

$$|x - \bigcirc(x)| < |x|\varepsilon_M.$$

The number $\varepsilon_M = \beta^{-(p-1)}$ is called the *machine epsilon*, and is a very useful quantity in numerical error analysis. It is the distance between 1.0 and the next larger floating point number. We will encounter ε_M on several occasions throughout this text.

Proof: Without loss of generality, we may assume that x is positive. If x happens to be a member of $\mathbb{F}$, then there is no rounding error, and the claim follows trivially. Thus we only need to consider the case $x \notin \mathbb{F}$. Since x is in the normalized range, it has a representation $x = (b_0.b_1b_2 \dots b_{p-1}b_p \dots)_\beta \times \beta^e$, where $b_0 \neq 0$. Using the fact that $x \notin \mathbb{F}$, it follows that its nearest neighbours in $\mathbb{F}$, $\bigtriangledown(x) = (b_0.b_1b_2 \dots b_{p-1})_\beta \times \beta^e$ and $\bigtriangleup(x) = [(b_0.b_1b_2 \dots b_{p-1})_\beta + \beta^{-(p-1)}] \times \beta^e$ are separated by the distance given by $\triangle(x) - \triangledown(x) = \beta^{-(p-1)} \times \beta^e = \varepsilon_M \beta^e$, and so $|x - \bigcirc(x)| < \varepsilon_M \beta^e$. Since x is normalized, we have that $x \geq 1 \times \beta^e$, so $|x - \bigcirc(x)| < x\varepsilon_M$. This gives the absolute error bound from which the relative bound follows immediately. Negative numbers are treated completely analogously. □

Note that in the case of rounding to nearest, the bounds of Theorem 1.3.5 can be decreased by a factor 0.5.

Exercise 1.3.6 *Show that the spacing between two adjacent floating point numbers x and y in the normalized range is bounded between $|x|\varepsilon_M/\beta$ and $|x|\varepsilon_M$.*

Example 1.3.7 *With base $\beta = 2$ and precision $p = 14$, the correctly rounded fraction $1/10$ is represented as:*

$$\triangledown(1/10) = (1.1001100110011)_2 \times 2^{-4} \qquad \triangle(1/10) = (1.1001100110100)_2 \times 2^{-4}.$$

Thus we have

$$\begin{aligned}\left|\frac{1/10 - \triangledown(1/10)}{1/10}\right| &= \left|\frac{(1.10011001100110011\ldots)_2 \times 2^{-4} - (1.1001100110011)_2 \times 2^{-4}}{(1.10011001100110011\ldots)_2 \times 2^{-4}}\right| \\ &= \left|\frac{(1.10011001100110011\ldots)_2 \times 2^{-20}}{(1.10011001100110011\ldots)_2 \times 2^{-4}}\right| = 2^{-16}.\end{aligned}$$

$$\begin{aligned}\left|\frac{1/10 - \triangle(1/10)}{1/10}\right| &= \left|\frac{(1.10011001100110011\ldots)_2 \times 2^{-4} - (1.1001100110100)_2 \times 2^{-4}}{(1.10011001100110011\ldots)_2 \times 2^{-4}}\right| \\ &= \left|\frac{(0.11001100110011001\ldots)_2 \times 2^{-15} - (1.0000000000000)_2 \times 2^{-15}}{(1.10011001100110011\ldots)_2 \times 2^{-4}}\right| \\ &< \left|\frac{(011)_2 \times 2^{-17} - (100)_2 \times 2^{-17}}{(1.100)_2 \times 2^{-4}}\right| = \left|\frac{3 \times 2^{-17} - 4 \times 2^{-17}}{(1.100)_2 \times 2^{-4}}\right| \\ &= \left|\frac{-1 \times 2^{-17}}{(1.100)_2 \times 2^{-4}}\right| = \frac{2}{3} \times 2^{-13}.\end{aligned}$$

Both relative errors are clearly bounded by $\varepsilon_M = 2^{-13}$.

It is important to keep in mind that Theorem 1.3.5 does *not* hold for subnormal numbers. In this situation, we can no longer use the fact that the leading digit b_0 in the floating point representation of x is non-zero. This prevents us from obtaining the desired bounds. Nevertheless, a simple modification of the proof of Theorem 1.3.5 gives the following error bounds:

Corollary 1.3.8 *If x is a real number such that $|x| = (b_0.b_1b_2\dots b_{p-1}b_p\dots)_\beta \times \beta^{\check{e}}$ with $b_i = 0$ for all $0 \le i < k \le p-1$ and $b_k \neq 0$, then the relative error caused by rounding to $\mathbb{F}_{\beta,p}^{\check{e},\hat{e}}$ is bounded by*

$$\left|\frac{x - \bigcirc(x)}{x}\right| < \beta^{-(p-1-k)}.$$

Equivalently, the corresponding absolute error is bounded by

$$|x - \bigcirc(x)| < |x|\beta^{-(p-1-k)}.$$

Thus, rounding in the subnormal range ($N_{min}^s \le |x| \le N_{max}^s$) leads to larger relative errors. The alternative, i.e., having no subnormal numbers at all, would result in flushing any number with $|x| < N_{min}^n$ to zero, which is of course even less desirable. Note that the requirement $k \le p-1$ ensures that $|x|$ is not smaller than the smallest positive subnormal number. Were this the case, the real number $|x|$ could be rounded *down* to zero, yielding a relative error of 1. It could also be rounded *up* to the smallest subnormal number, in which case the relative error could be arbitrarily large.

Example 1.3.9 *In the floating point system $\mathbb{F}_{2,10}^{-5,5}$, the correctly rounded real number 10^{-100} is represented as:*

$$\bigtriangledown(10^{-100}) = (0.000000000)_2 \times 2^{-5} \qquad \bigtriangleup(10^{-100}) = (0.000000001)_2 \times 2^{-5}.$$

Thus we have the relative error bounds:

$$\left|\frac{10^{-100} - \bigtriangledown(10^{-100})}{10^{-100}}\right| = \left|\frac{10^{-100} - (0.000000000)_2 \times 2^{-5}}{10^{-100}}\right| = 1.$$

$$\begin{aligned}\left|\frac{10^{-100} - \bigtriangleup(10^{-100})}{10^{-100}}\right| &= \left|\frac{10^{-100} - (0.000000001)_2 \times 2^{-5}}{10^{-100}}\right| \\ &= \left|\frac{10^{-100} - 2^{-14}}{10^{-100}}\right| > \frac{10^{-5}}{10^{-100}} = 10^{95}.\end{aligned}$$

1.4 Floating point arithmetic

The main mathematical concern about computing over a set of floating point numbers (i.e. a set $\mathbb{F}$ of type $\mathbb{F}_{\beta,p}^{\check{e},\hat{e}}$ or $\mathbb{F}_{\beta,p}$) is that it is not arithmetically closed. This means that if we take two floating point numbers $x, y \in \mathbb{F}$, and choose an arithmetic operator $\star \in \{+, -, \times, \div\}$ then, in general, the result will not be exactly representable in the floating point system: $x \star y \notin \mathbb{F}$. This is a property that is *not* shared by the real numbers $\mathbb{R}$, nor even the rationals $\mathbb{Q}$. No matter how high precision we use, this problem remains.

The only way we can define arithmetic on a set of floating point numbers $\mathbb{F}$ then is to associate the exact (in $\mathbb{R}$) outcome of a floating point operation with a representable floating point number. Naturally, this can be achieved by rounding the exact result from $\mathbb{R}$ to $\mathbb{F}$. Given any one of the arithmetic operations $\star \in \{+, -, \times, \div\}$, let $\circledast \in \{\oplus, \ominus, \otimes, \oslash\}$ denote the corresponding operation carried out in $\mathbb{F}$. We say that the floating point arithmetic is of maximum quality if

$$\text{(RG)} \qquad x, y \in \mathbb{F} \text{ and } \star \in \{+, -, \times, \div\} \Rightarrow x \circledast y = \bigcirc(x \star y).$$

Property (RG) states that floating point arithmetic should yield the same result as if though the computation was carried out with infinite precision, after which the exact result is rounded to the appropriate neighbouring floating point. Thus the only error is incurred by the final rounding to $\mathbb{F}$, and consequently, the result of any arithmetic operation has the same quality as the rounding itself. It is true, but not obvious, that this can be practically implemented[6].

Theorem 1.4.1 *Let $\star$ denote one of the arithmetic operators $+, -, \times, \div$. Then, if x and y are normal floating point numbers with $x \star y \neq 0$, the relative error of the floating point operation is bounded by*

$$\left| \frac{x \star y - x \circledast y}{x \star y} \right| < \varepsilon_M.$$

Equivalently, the corresponding absolute error is bounded by

$$|x \star y - x \circledast y| < |x \star y| \varepsilon_M.$$

It is important to realize that Theorem 1.4.1 is only valid for *one single* floating point operation. All bets are off when several operations are involved. An expression like

$$f(x) = 1 \oslash ((1 \oplus x) \ominus 1)$$

may return a grossly incorrect value when $|x| < \varepsilon_M$, depending on the rounding mode. The following example serves as an illustration to how these types of inaccuracies may seriously affect the outcome of a simple numerical experiment.

Example 1.4.2 *Consider the ternary shift map $f\colon [0, 1] \to [0, 1]$ defined by $f(x) = 3x \bmod 1$. This map has a period-4 cycle $\frac{1}{10} \to \frac{3}{10} \to \frac{9}{10} \to \frac{7}{10} \to \frac{1}{10}$. Starting a*

[6] In fact, it suffices to use registers having $p + 2$ digits of precision, combined with a so called *sticky bit* to obtain maximum quality in the arithmetic operations, see e.g. [Go91], [Kn98], or [Ko02].

numerical iteration (over $\mathbb{F}_{2,53}$) at $x_0 = \frac{1}{10}$, however, produces the following orbit:

$$\begin{aligned} x(0) &= 0.10000000000000001 & x(1) &= 0.30000000000000004\\ x(2) &= 0.90000000000000013 & x(3) &= 0.70000000000000018\\ x(4) &= 0.10000000000000053 & x(5) &= 0.3000000000000016\\ &\vdots & & \\ x(47) &= 0.15273362128584456 & x(48) &= 0.45820086385753367\\ x(49) &= 0.37460259157260101 & x(50) &= 0.12380777471780302\\ x(51) &= 0.37142332415340906 & x(52) &= 0.11426997246022719 \end{aligned}$$

After less than 50 *iterates, there is no sign of the periodic orbit! The reason is that the function f is expanding, i.e., its derivate (when defined) is greater than one everywhere. As a consequence, even very small initial errors will eventually be grossly magnified, and completely saturate the computed orbit. This intrinsic property of expanding maps makes the numerical study of chaotic dynamical systems a very challenging topic.*

Another consequence of computing with finite precision is that many basic mathematical laws no longer hold. For example, addition and multiplication are no longer associative.

Example 1.4.3 *Consider the numbers $x = 1.234 \times 10^4$, $y = -1.235 \times 10^4$, and $z = 1.002 \times 10^1$, all belonging to $\mathbb{F}_{10,4}^{-9,9}$. Rounding to nearest even, we have*

$$(x \bigoplus y) \bigoplus z = -1.000 \times 10^1 \bigoplus 1.002 \times 10^1 = 2.000 \times 10^{-2},$$

whereas

$$x \bigoplus (y \bigoplus z) = 1.234 \times 10^4 \bigominus 1.234 \times 10^4 = 0.000 \times 10^{-9}.$$

The first result is exact, while the second suffers from inaccuracies caused by a lack of precision.

Exercise 1.4.4 *How many pairs of floating point numbers can be exactly added in the set $\mathbb{F}_{2,3}^{-1,2}$? How many pairs can not? What about the general case $\mathbb{F}_{\beta,p}^{\check{e},\hat{e}}$?*

Exercise 1.4.5 *Assuming the property (RG), show that the following statements always hold true:*

(1) $x \in \mathbb{F} \Rightarrow 1 \bigotimes x = x$;

(2) $x \in \mathbb{F} \setminus \{-\infty, 0, +\infty\} \Rightarrow x \bigoplus\!\!\!\!\div x = 1$;

(3) $x \in \mathbb{F}$ *(with $\beta = 2$)* $\Rightarrow 0.5 \bigotimes x = x \bigoplus\!\!\!\!\div 2$.

(4) $x, y \in \mathbb{F} \Rightarrow (x \ominus y = 0) \Rightarrow (x = y)$.

Also show that statement (4) is false if $\mathbb{F}$ *has no subnormal numbers.*

In view of (RG), we can give a computational definition of the machine epsilon. Even if the base and precision of the underlying floating point system are unknown, this definition allows for a direct computation of ε_M.

Definition 1.4.6 *In a floating point system with base* β *and precision* p*, we call the number* $\varepsilon_M = \beta^{-(p-1)}$ *the machine epsilon. It is the smallest positive floating point number* x *that satisfies* $1 < 1 \oplus x$ *when rounding down, i.e.,* $\varepsilon_M = \min\{x \in \mathbb{F} \colon 1 < \triangledown(1 \oplus x)\}$.

Note that the condition $1 < \triangledown(1 \oplus x)$ is very different from the *seemingly* equivalent condition $0 < \triangledown(x)$. Some aggressively optimizing compilers do not realize this distinction, and thus produce grossly incorrect code. The smallest floating point number x satisfying $0 < \triangledown(x)$ is called the *machine eta*, and is denoted by η_M. This is the smallest positive subnormal number, and is thus equal to $\beta^{\check{e}-p+1} = \beta^{\check{e}}\varepsilon_M$. A program that computes both ε_M and η_M is presented in Appendix B.1. These computations can safely be carried out in the default rounding mode, round to nearest even, since the least significant bits of both 1.0 and 0.0 is zero, which is an even number.

1.5 The IEEE standard

In the early days of computing, each brand of computer had its own implementation for floating point operations. This had the unfortunate effect that the outcome of a computation heavily depended on the precise type of machine used to perform the calculations. Naturally, this also severely limited the portability of programs, as code that worked perfectly well on one machine could crash on another. Even worse, most computer manufacturers had their own internal *representation* of floating point numbers. Of course, this led to the fact that data transfer between different machines became a highly complex task.

In the second half of the eighties, an international standard for the use and representation of floating point numbers was agreed upon. The standard is actually two standards: the IEEE p754, released in 1985, which deals exclusively with binary representations, and the IEEE p854, released in 1987, which in a common framework considers both base 2 and 10, see [IE85] and [IE87], respectively. Besides demanding maximal quality[7] of the arithmetic of floating point numbers, the standard also

[7] The IEEE standard requires that the basic operations $\{+, -, \times, \div\}$ and $\surd$ return the floating point nearest to the exact result, with regards to the rounding mode. Rather surprisingly, no such demands are imposed on the trigonometric and exponential functions. On a HP 9000/700, the argument $x = 2.50 \times 10^{17}$ produces the grossly incorrect $\sin x = 4.14415 \times 10^{7}$.

requires a consistent treatment of exceptions (e.g. division by zero) which greatly facilitates the construction of robust code. The IEEE standard also requires the presence of the four basic rounding modes: round up, round down, round to zero, and round to nearest (even).

For two very nice expositions of the IEEE floating point standard, see [Go91] and [Ov01].

1.5.1 The IEEE formats

The IEEE standard specifies two basic types of floating point numbers: the `single` and `double` formats. Extended versions of these formats are also mentioned, although only minimal requirements (as opposed to exact descriptions) for these are specified. Whereas the extended double format is provided on most architectures, the extended single format has found little support. We will therefore only cover the extended double format, which henceforth is denoted `extended`.

The standard also introduces three special symbols: `-Inf`, `+Inf`, and `NaN`. The two first are direct analogues to the mathematical notion of $\pm\infty$. Any finite floating point number x satisfies `-Inf` $< x <$`+Inf`. A real number greater that the largest finite floating point number is represented as either N^n_{max} or `+Inf`, depending on the rounding mode. Similarly, a real number smaller than the smallest finite floating point number is represented as either $-N^n_{max}$ or `-Inf`. The symbol `NaN`, which is short for *not a number*, is returned whenever the outcome of a floating point operation is undefined, e.g. $0/0$ or `Inf` $-$ `Inf`.

To simplify the exposition, in all that follows, we will consider only floating point representations in binary base. Although this base has some special advantages, the theory presented can easily be generalized to a setting with an arbitrary base.

On almost all commercial computers, the two basic formats `single` and `double` are implemented using exactly the same mould, varying only two parameters. Each format is made up of a fixed number $F_\star$ of bits (binary integers), where the subscript $\star \in \{s, d\}$ indicates the specific format in question. The first bit encodes the sign of the floating point number, the following $E_\star$ bits correspond to the exponent, and the remaining $M_\star$ bits represent the mantissa. Thus, any two elements of $\{F_\star, E_\star, M_\star\}$ completely define the format in question. Actually, each format has precision $M_\star + 1$. This is achieved by a *hidden bit*: since a normal floating point number represented in base two must start with a one, there is no need to explicitly store this leading bit. Note that this trick only works for binary representations. The exponent also has a little twist to it: we are not storing the actual exponent, but rather a *biased* version of it. Using $E_\star$ bits, we can represent any integer between 0 and $2^{E_\star} - 1$. We form the actual exponent by subtracting the bias $B_\star = 2^{E_\star - 1} - 1$ from the stored number. This gives a exponent range of $[-2^{E_\star - 1} + 1, 2^{E_\star - 1}]$. The two boundary points, however, are reserved for non-normal numbers.

1	$E_\star$	$M_\star$
σ	e	m

Figure 1.6: The basic IEEE formats.

Consider the $F_\star$-bit string $[\sigma; e_1e_2\dots e_{E_\star}; m_1m_2\dots m_{M_\star}]$, and let $E = (e_1e_2\dots e_{E_\star})_2$ and $M = (0.m_1m_2\dots m_{M_\star})_2$. Then the floating point number x represented by the string is decoded as follows:

(a) if $E = 2^{E_\star} - 1$ and $M \neq 0$, then $x = \texttt{NaN}$;

(b) if $E = 2^{E_\star} - 1$ and $M = 0$, then $x = (-1)^\sigma\texttt{Inf}$;

(c) if $0 < E < 2^{E_\star} - 1$, then $x = (-1)^\sigma 1.M \times 2^{E-B_\star}$;

(d) if $E = 0$ and $M \neq 0$, then $x = (-1)^\sigma 0.M \times 2^{1-B_\star}$;

(e) if $E = 0$ and $M = 0$, then $x = (-1)^\sigma \times 0$.

We see that case (c) corresponds to the set of normal numbers, whereas case (d) deals with the subnormal numbers. Note that cases (d) and (e) could be merged, although we chose not to do so seeing that zero is not a subnormal number.

The `single` format consists of 32 bits, of which eight bits correspond to the exponent, and the remaining 23 bits represent the mantissa. In other words, we have $\{F_s, E_s, M_s\} = \{32, 8, 23\}$. The smallest and largest positive normal numbers in `single` format are seen to be $N^n_{min} = 2^{-126} \approx 1.2 \times 10^{-38}$ and $N^n_{max} = (2 - 2^{-23}) \times 2^{127} \approx 2^{128} \approx 3.4 \times 10^{38}$, respectively. The machine epsilon is $\varepsilon_M = 2^{-23} \approx 1.2 \times 10^{-7}$. This means that we should expect about seven significant decimal digits.

1	8	23
σ	e	m

Figure 1.7: The IEEE `single` format.

The `double` format consists of 64 bits, of which 11 bits correspond to the exponent, and the remaining 52 bits represent the mantissa, i.e., we have $\{F_d, E_d, M_d\} = \{64, 11, 52\}$. The smallest and largest positive normal numbers in `double` format are seen to be $N^n_{min} = 2^{-1022} \approx 2.2 \times 10^{-308}$ and $N^n_{max} = (2 - 2^{-52}) \times 2^{1023} \approx 2^{1024} \approx 1.8 \times 10^{308}$, respectively. The machine epsilon is $\varepsilon_M = 2^{-52} \approx 2.2 \times 10^{-16}$, so we can expect about 16 significant decimal digits from the `double` format.

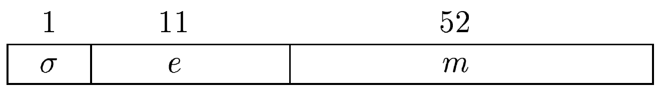

Figure 1.8: The IEEE `double` format.

1.5.2 The `extended` format

In contrast to the `single` and `double` formats, the IEEE standard does not specify absolute parameters for the `extended` format. Instead, minimal requirements are given, and it is up to each individual manufacturer to decide the precise parameters to be used. The requirements for the three formats are illustrated below.

Table 1.1: The most common IEEE formats.

	`single`	`double`	`extended`
Format width in bits	32	64	≥ 79
Exponent width in bits	8	11	≥ 15
Precision p	24	53	≥ 64
Exponent bias	+127	+1023	unspecified
Maximal exponent	+127	+1023	$\geq +16383$
Minimal exponent	-126	-1022	≤ -16382

Considering that the `extended` format is not precisely specified, it is rather unfortunate that it has become the most commonly used format. To make matters worse, the non-expert user is often unaware of this fact. The reason for this state of affairs is that almost all computers perform intermediate computations in the widest registers available to them. Even the simplest computation, involving only two `double` type variables, will be converted to and performed in `extended` format, after which the result is rounded back to the `double` format. This can (and often does!) lead to quite unexpected results, which are very hard to "debug", see Example 1.6.1. In light of this, we will spend quite some time describing the various "flavours" of the `extended` format.

Let us begin with the 128-bit `extended` format, which is provided on the SPARC architecture. This format follows the generic mould described in the previous section, with parameters $\{F_e, E_e, M_e\} = \{128, 15, 112\}$. Thus, the smallest and largest positive normal numbers in the 128-bit `extended` format are seen to be $N^n_{min} = 2^{-16382} \approx 3.4\times10^{-4932}$ and $N^n_{max} = (2-2^{-112})\times2^{16383} \approx 2^{16384} \approx 1.2\times10^{4932}$, respectively. The machine epsilon is $\varepsilon_M = 2^{-112} \approx 1.9 \times 10^{-34}$. This means that we should expect about 34 significant decimal digits from the 128-bit `extended` format.

In contrast to SPARC, the Intel x86 and Pentium architectures provide an 80-bit

1	15	112
σ	e	m

Figure 1.9: The SPARC 128-bit `extended` format.

`extended` format, made up by ten 8-bit bytes[8]. This format consists of a 1-bit sign, a 15-bit (biased) exponent, and a 64-bit mantissa. In the 64-bit mantissa no hidden bit is employed, which leads to some minor peculiarities.

1	15	64
σ	e	m

Figure 1.10: The Intel 80-bit `extended` format.

Consider the 80-bit string $[\sigma; e_1e_2\dots e_{15}; m_0m_1m_2\dots m_{63}]$. Let $E = (e_1e_2\dots e_{15})_2$, and $M = (0.m_1m_2\dots m_{63})_2$. Then the floating point number x represented by the string is decoded as follows:

(a) if $m_0 = 1$, $E = 32767$, and $M \neq 0$, then $x =$ `NaN`;

(b) if $m_0 = 1$, $E = 32767$, and $M = 0$, then $x = (-1)^\sigma$`Inf`;

(c) if $m_0 = 1$ and $0 < E < 32767$, then $x = (-1)^\sigma 1.M \times 2^{E-16383}$;

(d) if $m_0 = 0$, $E = 0$, and $M \neq 0$, then $x = (-1)^\sigma 0.M \times 2^{-16382}$;

(e) if $m_0 = 0$, $E = 0$, and $M = 0$, then $x = (-1)^\sigma \times 0$;

(f) if $m_0 = 0$ and $0 < E < 32767$, then x is not defined;

(g) if $m_0 = 1$ and $E = 0$, then $x = (-1)^\sigma 1.M \times 2^{-16382}$;

The numbers corresponding to case (g) are called *pseudo-subnormal numbers*. These are never generated as results, but may appear as operands, in which case they are implicitly converted to the corresponding normal numbers as in (c).

The smallest and largest positive normal numbers in the 80-bit `extended` format are seen to be $N^n_{min} = 2^{-16382} \approx 3.4 \times 10^{-4932}$ and $N^n_{max} = (2 - 2^{-63}) \times 2^{16383} \approx 2^{16384} \approx 1.2 \times 10^{4932}$, respectively. The machine epsilon is $\varepsilon_M = 2^{-63} \approx 1.1 \times 10^{-19}$. This means that we should expect about 19 significant decimal digits from the 80-bit `extended` format.

[8]Under the UNIX System V operating system, however, the format is made up by three 32-bit words, leaving the 16 highest addressed bits unused.

The IBM S/390 G5 series, which uses a hexadecimal base, has hardware support for the IEEE formats with parameters matching those of the SPARC, see [SK99]. The CRAY T90 series has also moved toward the IEEE formats, but with some important exceptions. First, there is a slight linguistic discrepancy: the CRAY `single` format is 64 bits wide, and thus corresponds to the IEEE `double`. Similarly, the CRAY `double` format is 128 bits wide, and therefore corresponds to the IEEE `extended`. What is more serious is that the CRAY architecture does not treat subnormal numbers according to the IEEE standard. In fact, all subnormal numbers are forced to zero, see [Ga96]. As we have seen, this leads to several important mathematical laws being violated. To further confuse matters, the Macintosh PowerPC Numerics Environment provides a *double-double* format that is 128 bits wide. The exponent field, however, is only 11-bits wide, which is lower than the requirement for an IEEE `extended` format. The *double-double* is implemented in software combining two `double` formats in a quite complicated manner.

We end this section by listing some important facts about the various formats. Here, "m-bits" denotes the number of bits reserved for the mantissa, and "e-bits" corresponds to the exponent field.

Table 1.2: Summary of the most common IEEE formats.

format	bits	m-bits	e-bits	N^s_{min}	N^n_{max}
`single`	32	23+1	8	1.4×10^{-45}	3.4×10^{38}
`double`	64	52+1	11	4.9×10^{-324}	1.8×10^{308}
INTEL `extended`	80	64	15	3.6×10^{-4951}	1.2×10^{4932}
SPARC `extended`	128	112+1	15	6.5×10^{-4966}	1.2×10^{4932}

1.6 Examples of floating point computations

A common way to determine the accuracy a specific computation is to gradually increase the precision until the result stabilizes. If adding more bits to the floating point representation does not alter the result of the computation, one usually accepts the result as being correct. The following example illustrates the false sense of security given by this approach.

Example 1.6.1 *Consider the function*

$$f(x, y) = 333.75y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + 5.5y^8 + x/(2y).$$

As observed in [Ru88], using FORTRAN on an IBM S/370 ($\beta = 16$), the function evaluated at the point $(\tilde{x}, \tilde{y}) = (77617, 33096)$ produces the following output:

type	*p*	$f(\tilde{x}, \tilde{y})$
`REAL*4`	*24*	1.172603...
`REAL*8`	*53*	1.1726039400531...
`REAL*10`	*64*	1.172603940053178...

Using C or C++ (with gcc/g++) on an Intel Pentium III chip ($\beta = 2$), we get

type	*p*	$f(\tilde{x}, \tilde{y})$
`float`	*24*	178702833214061281280
`double`	*53*	178702833214061281280
`long double`	*64*	178702833214061281280

Using C or C++ (with gcc/g++) on a Sun UltraSPARC ($\beta = 2$), we get

type	*p*	$f(\tilde{x}, \tilde{y})$
`float`	*24*	2571784163840789082227689390O8
`double`	*53*	1.17260394005317863494924440605̈9...
`long double`	*113*	1.17260394005317863494924440605̈9...

Although all coefficients are exactly representable in base 2 (and thus in base 16), the rounding errors render the result useless. The correct answer is actually $-0.8273960599\ldots$*, which means that we did not even get the sign right! These discrepancies are due to the fact that the two terms* $T_1 = 5.5\tilde{y}^8$ *and* $T_2 = 333.75\tilde{y}^6 + \tilde{x}^2(11\tilde{x}^2\tilde{y}^2 - \tilde{y}^6 - 121\tilde{y}^4 - 2)$ *are very large in modulus, and almost cancel:*

$$T_1 = +7917111340668961361101134701524942848$$
$$T_2 = -7917111340668961361101134701524942850.$$

Since the sum of these terms is $T_1 + T_2 = -2$*, we are left with just*

$$f(\tilde{x}, \tilde{y}) = T_1 + T_2 + \tilde{x}/(2\tilde{y}) = -2 + \tilde{x}/(2\tilde{y}),$$

which gives

$$f(\tilde{x}, \tilde{y}) = -2 + \frac{77617}{2 \times 33096} \approx -0.8273960599.$$

Of course, when computing with any of the above-mentioned floating points formats, we have cancellation, and the sum of the two huge terms (both of magnitude $\approx 7.9 \times 10^{36}$*) is evaluated as zero. This results in the approximate function value*

$$f(\tilde{x}, \tilde{y}) = 0 + \frac{77617}{2 \times 33096} \approx 1.1726039400531.$$

This, however, does not explain the results from the Intel Pentium III chip. In this case, with no explicit instructions passed on to the compiler, all intermediate results are converted and worked upon in the `long double` *format by default.*

For a more thorough analysis of this example, see [EW02] and [CV01].

The next example deals with the (seemingly) simple task of generating the graph of a polynomial.

Example 1.6.2 *Plot the graph, and search for roots of the polynomial*

$$p(t) = t^6 - 6t^5 + 15t^4 - 20t^3 + 15t^2 - 6t + 1.$$

MATLAB produces the non-smooth graph illustrated in Figure 1.11. This picture is

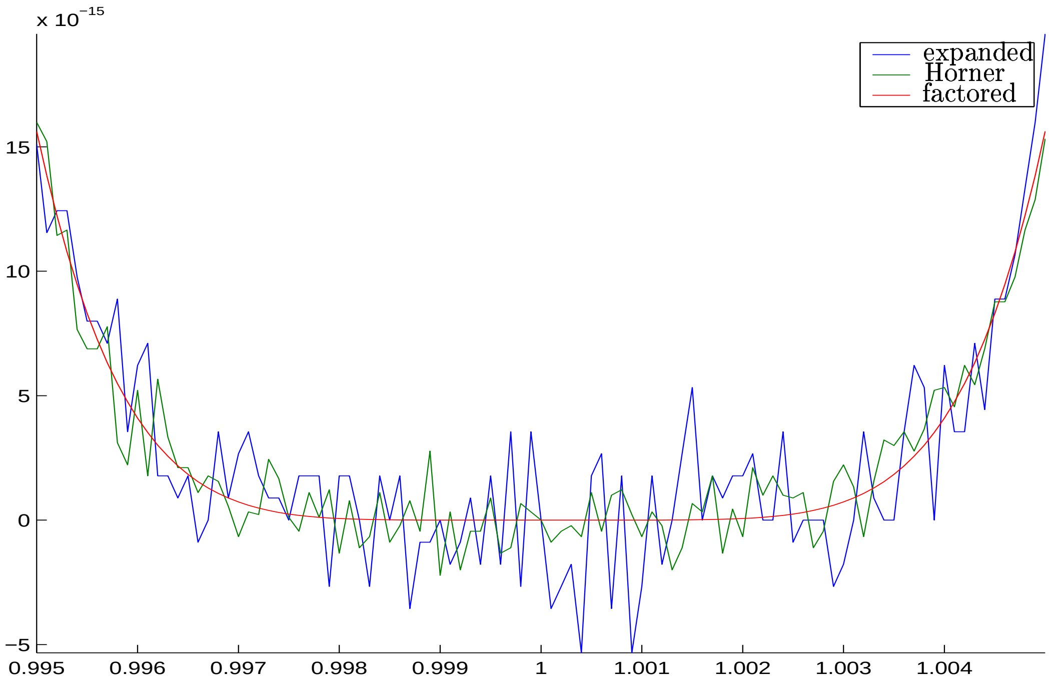

Figure 1.11: A smooth graph of a polynomial?

clearly wrong: a polynomial of degree n can have at most $n-1$ local extrema. Even if we minimize the number of floating point operations by evaluating the polynomial via Horner's method

$$p(t) = ((((((t-6)t+15)t-20)t+15)t-6)t+1)$$

the resulting graph is clearly not correct. Note, however, that we can factor the polynomial as $p(t) = (t-1)^6$, which is (correctly) plotted as the smooth graph in Figure 1.11. It follows that the graph should lie above the t-axis, except at the multiple root $t^* = 1$.

The reason why the computed values approximate the graph so poorly is that the condition number *of $p(t)$ near $t^* = 1$ is very large. Without going into explicit calculations, the condition number of p near t^* is given by*

$$\kappa(t) \approx 6\left|\frac{t}{t-t^*}\right|.$$

As t approaches the multiple root at t^*, we see that κ tends to $+\infty$. A large condition numbers translates to poor accuracy, as is illustrated in Figure 1.11. Note that this situation would not occur if the multiple root was positioned at 0 *rather than* 1.

The conclusion we draw is that function evaluations depend on the function *representation* when computed over $\mathbb{F}$. This is a difficult fact to accept for most mathematicians, who are used to computing over $\mathbb{R}$.

After these unnerving examples, let us show how one can obtain rigorous mathematical statements using the computer. By utilizing the directed rounding modes, we can enclose of the exact result of certain computations. These techniques will be generalized and studied in detail in Chapter 2.

Example 1.6.3 *It is well-known that the infinite series $S = \sum_{k=1}^{\infty} k^{-2}$ has the exact value $\pi^2/6$. Assume for the moment that we are unaware of this, and suppose that we need to find an approximation to S, say to 12 decimal places. Clearly, we cannot sum an infinite number of terms on the computer, so let us split the series into two pieces:*

$$S = \sum_{k=1}^{\infty} k^{-2} = \sum_{k=1}^{N} k^{-2} + \sum_{k=N+1}^{\infty} k^{-2} = S_N + S_N^\star.$$

Our strategy is now to bound the infinite part $S_N^\star$ by mathematical means, whereas we calculate the finite part S_N using the computer.

In order to achieve 12 *correct decimal places, we must ensure that the upper and lower bounds for $S_N^\star$ differ by at most 5×10^{-13}. By a simple geometric argument (draw the picture!), we have*

$$\int_{N+1}^{\infty} \frac{dx}{x^2} < S_N^\star < \int_{N+1}^{\infty} \frac{dx}{(x-1)^2},$$

which produces the bounds $\frac{1}{N+1} < S_N^\star < \frac{1}{N}$ of width $\delta_N = \frac{1}{N(N+1)}$. Taking $N = 2\times 10^6$ gives $\delta_N < 2.5 \times 10^{-13}$, which should do nicely, assuming that we can compute the finite part S_N accurately.

So let $N = 2 \times 10^6$, and compute the sum $S_N = \sum_{k=1}^{N} k^{-2}$ with `double` *precision, using the directed rounding modes. This produces the following output:*

```
Rounded down: S_N = 1.644 933 566 626 364 25
Rounded up  : S_N = 1.644 933 567 070 448 80.
```

As is plain to see, the results differ already in the 9th decimal. Since all terms are positive, however, the IEEE standard guarantees that the exact result is bounded from below by the result obtained when always rounding down. Analogously, the *exact result is bounded from above by the result obtained when always rounding up. Thus we can enclose the exact value of S_N in the interval*

$$[1.64493356662636425, 1.64493356707044880] \stackrel{\text{def}}{=} 1.64493356^{707044880}_{662636425}.$$

As the number of terms to be summed is known in advance, we can get better accuracy by adding the terms in increasing order (can you explain why?). Doing so yields the results

```
Rounded down: S_N = 1.644 933 566 848 350 46
Rounded up  : S_N = 1.644 933 566 848 352 46,
```

which now differ in the 15th decimal. Once again, we know that the exact value of S_N is contained in the interval

$$[1.64493356684835046, 1.64493356684835246] = 1.64493356684835^{246}_{046},$$

which allows us to refine our numerical enclosure of the partial sum:

$$S_N \in 1.64493356684835^{246}_{046} \subset 1.64493356^{707044880}_{662636425}.$$

Combining this information with the fact that $\frac{1}{N+1} < S_N^\star < \frac{1}{N}$ produces the final enclosure:

$$S \in 1.644934066848^{35253}_{10028},$$

which is correct to 12 decimal places. Compare this to the "exact" value $S = \pi^2/6 \approx$ 1.64493406684822630, where we have approximated π by its 50 leading digits.

It is exactly this mixture of mathematics and properly rounded numerical computations that opens the door to *validated numerics*. As we have seen, these techniques allow us to construct computer-aided mathematical *proofs*, just like our recent proof that all digits of the approximation $S \approx 1.644934066848$ are correct.