Chapter 2
Roundoff Errors

As observed in Chapter 1, various errors may arise in the process of calculating an approximate solution for a mathematical model. In this chapter we introduce and discuss in detail the most fundamental source of imperfection in numerical computing: roundoff errors. Such errors arise due to the intrinsic limitation of the finite precision representation of numbers (except for a restricted set of integers) in computers.

Different audiences may well require different levels of depth and detail in the present topic. We therefore start our discussion in Section 2.1 with the bare bones: a collection of essential facts related to floating point systems and roundoff errors that may be particularly useful for those wishing to concentrate on the last seven chapters of this text.

Note: If you do not require detailed knowledge of roundoff errors and their propagation during a computation, not even the essentials of Section 2.1, then you may skip this chapter (not recommended), at least upon first reading. What you must accept, then, is the notion that each number representation and each elementary operation (such as $+$ or $*$) in standard floating point arithmetic introduces a small, random relative error: up to about 10^{-16} in today's standard floating point systems.

In Section 2.2 we get technical and dive into the gory details of floating point systems and floating point arithmetic. Several issues only mentioned in Section 2.1 are explained here.

The small representation errors as well as errors that arise upon carrying out elementary operations such as addition and multiplication are typically harmless unless they accumulate or get magnified in an unfortunate way during the course of a possibly long calculation. We discuss roundoff error accumulation as well as ways to avoid or reduce such effects in Section 2.3.

Finally, in Section 2.4, the IEEE standard for floating point arithmetic, which is implemented in any nonspecialized hardware, is briefly described.

2.1 The essentials

This section summarizes what we believe all our students should know as a minimum about floating point arithmetic. Let us start with a motivating example.

Example 2.1. Scientists and engineers often wish to believe that the numerical results of a computer calculation, especially those obtained as output of a software package, contain no error—at least not a significant or intolerable one. But careless numerical computing does occasionally lead to trouble.

Note: The word "essential" is not synonymous with "easy." If you find some part of the description below too terse for comfort, then please refer to the relevant section in this chapter for more motivation, detail, and explanation.

One of the more spectacular disasters was the Patriot missile failure in Dhahran, Saudi Arabia, on February 25, 1991, which resulted in 28 deaths. This failure was ultimately traced to poor handling of roundoff errors in the missile's software. The webpage http://www.ima.umn.edu/~arnold/disasters/patriot.html contains the details of this story. For a large collection of software bugs, see http://wwwzenger.informatik.tu-muenchen.de/persons/huckle/bugse.html. ∎

Computer representation of real numbers

Computer memory has a finite capacity. This obvious fact has far-reaching implications on the representation of real numbers, which in general *do not* have a finite uniform representation. How should we then represent a real number on the computer in a way that can be sensibly implemented in hardware?

Any real number $x \in \mathcal{R}$ is accurately representable by an infinite sequence of digits.[4] Thus, we can write

$$x = \pm(1.d_1 d_2 d_3 \cdots d_{t-1} d_t d_{t+1} \cdots) \times 2^e,$$

where e is an integer *exponent*. The (possibly infinite) set of binary digits $\{d_i\}$ each have a value 0 or 1. The decimal value of this *mantissa* is

$$1.d_1 d_2 d_3 \cdots = 1 + \frac{d_1}{2} + \frac{d_2}{2^2} + \frac{d_3}{2^3} + \cdots.$$

For instance, the binary representation $x = -(1.10100\cdots) \times 2^1$ has in decimal the value

$$x = -(1 + 1/2 + 1/8) \times 2 = -3.25.$$

Of course, it should be clear that the choice of a binary representation is just one of many possibilities, indeed a convenient choice when it comes to computers. To represent any real number on the computer, we associate to x a *floating point representation* $\mathrm{fl}(x)$ of a form similar to that of x but with only t digits, so

$$\mathrm{fl}(x) = \mathrm{sign}(x) \times (1.\tilde{d}_1 \tilde{d}_2 \tilde{d}_3 \cdots \tilde{d}_{t-1} \tilde{d}_t) \times 2^e$$

for some t that is fixed in advance and binary digits $\tilde{d}_i$ that relate to d_i in a manner that will soon be specified. Storing this fraction in memory thus requires t bits. The exponent e must also be stored in a fixed number of bits and therefore must be bounded, say, between a lower bound L and an upper bound U. Further details are given in Section 2.2, which we hope will provide you with much of what you'd like to know.

Rounding unit and standard floating point system

Without any further details it should already be clear that representing x by $\mathrm{fl}(x)$ necessarily causes an error. A central question is how accurate the floating point representation of the real numbers is.

[4]It is known from calculus that the set of all rational numbers in a given real interval is dense in that interval. This means that any number in the interval, rational or not, can be approached to arbitrary accuracy by a sequence of rational numbers.

Specifically, we ask how large the relative error in such a representation, defined by

$$\frac{|\mathrm{fl}(x) - x|}{|x|},$$

can be.

Modern floating point systems, such as the celebrated IEEE standard described in detail in Section 2.4, guarantee that this relative error is bounded by

$$\eta = \frac{1}{2} \times 2^{-t}.$$

The important quantity η, which will be more generally defined in Section 2.2, has been called **rounding unit**, **machine precision**, **machine epsilon**, and more. We will use the term rounding unit throughout.

The usual floating point word in the standard floating point system, which is what MATLAB uses by default, has 64 bits. Of these, 52 bits are devoted to the mantissa (or fraction) while the rest store the sign and the exponent. Hence the rounding unit is

$$\eta = 2^{-53} \approx 1.1 \times 10^{-16}.$$

This is called *double precision*; see the schematics in Figure 2.1.

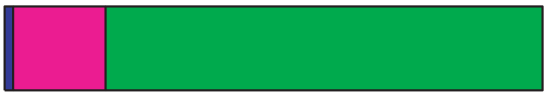

Figure 2.1. *A double word (64 bits) in the standard floating point system. The blue bit is for sign, the magenta bits store the exponent, and the green bits are for the fraction.*

If the latter name makes you feel a bit like starting house construction from the second floor, then rest assured that there is also a *single precision* word, occupying 32 bits. This one obviously has a much smaller number of digits t, hence a significantly larger rounding unit η. We will not use single precision for calculations anywhere in this book except for Examples 2.2 and 14.6, and neither should you.

Roundoff error accumulation

Even if number representations were exact in our floating point system, *arithmetic operations* involving such numbers introduce roundoff errors. These errors can be quite large in the relative sense, unless *guard digits* are used. These are extra digits that are used in interim calculations. The IEEE standard requires **exact rounding**, which yields that the relative error in each arithmetic operation is also bounded by η.

Given the above soothing words about errors remaining small after representing a number and performing an arithmetic operation, can we really put our minds at ease and count on a long and intense calculation to be as accurate as we want it to be?

Not quite! We have already seen in Example 1.3 that unpleasant surprises may arise. Let us mention a few potential traps. The fuller version is in Section 2.3.

Careless computations can sometimes result in division by 0 or another form of undefined numerical result. The corresponding variable name is then assigned the infamous designation `NaN`. This is a combination of letters that stands for "not a number," which naturally one dreads to see in one's own calculations, but it allows software to detect problematic situations, such as an attempt to divide 0 by 0, and do something graceful instead of just halting. We have a hunch that you will inadvertently encounter a few `NaN`'s before you finish implementing all the algorithms in this book.

There is also a potential for an *overflow*, which occurs when a number is larger than the largest that can be represented in the floating point system. Occasionally this can be avoided, as in Example 2.9 and other examples in Section 2.3.

We have already mentioned in Section 1.3 that a roundoff error accumulation that grows *linearly* with the number of operations in a calculation is inevitable. Our calculations will never be so long that this sort of error would surface as a practical issue. Still, there are more pitfalls to watch out for. One painful type of error magnification is a *cancellation error*, which occurs when two nearly equal numbers are subtracted from one another. There are several examples of this in Section 2.3. Furthermore, the discussion in this chapter suggests that such errors may consistently arise in practice.

The rough appearance of roundoff error

Consider a smooth function g, sampled at some t and at $t+h$ for a small (i.e., near 0) value h. Continuity then implies that the values $g(t)$ and $g(t+h)$ are close to each other. But the rounding errors in the machine representation of these two numbers are unrelated to each other: they are random for all we know! These rounding errors are both small (that's what η is for), but even their signs may in fact differ. So when subtracting $g(t+h)-g(t)$, for instance, on our way to estimate the derivative of g at t, the relative roundoff error becomes significantly larger as cancellation error naturally arises. This is apparent already in Figure 1.3.

Let us take a further look at the seemingly unstructured behavior of roundoff error as a smooth function is being sampled.

Example 2.2. We evaluate $g(t)=e^{-t}(\sin(2\pi t)+2)$ at 501 equidistant points between 0 and 1, using the usual double precision as well as *single precision* and plotting the differences. This essentially gives the rounding error in the less accurate single precision evaluations. The following MATLAB instructions do the job:

```
t = 0:.002:1;
tt = exp(-t) .* (sin(2*pi*t)+2);
rt = single(tt);
round_err = (tt - rt) ./tt ;
plot (t,round_err,'b-');
title ('error in sampling exp(-t)(sin(2\pi t)+2) single precision')
xlabel('t')
ylabel('roundoff error')

% relative error should be about eta = eps(single)/2
rel_round_err = max(abs(round_err)) / (eps('single')/2)
```

Thus, the definition of the array values `tt` is automatically implemented in double precision, while the instruction `single` when defining the array `rt` records the corresponding values in single precision.

The resulting plot is depicted in Figure 2.2. Note the disorderly, "high frequency" oscillation of the roundoff error. This is in marked contrast to discretization errors, which are usually "smooth," as we have seen in Example 1.2. (Recall the straight line drop of the error in Figure 1.3 for relatively large h, which is where the discretization error dominates.)

The output of this program indicates that, as expected, the relative error is at about the rounding unit level. The latter is obtained by the (admittedly unappealing) function call

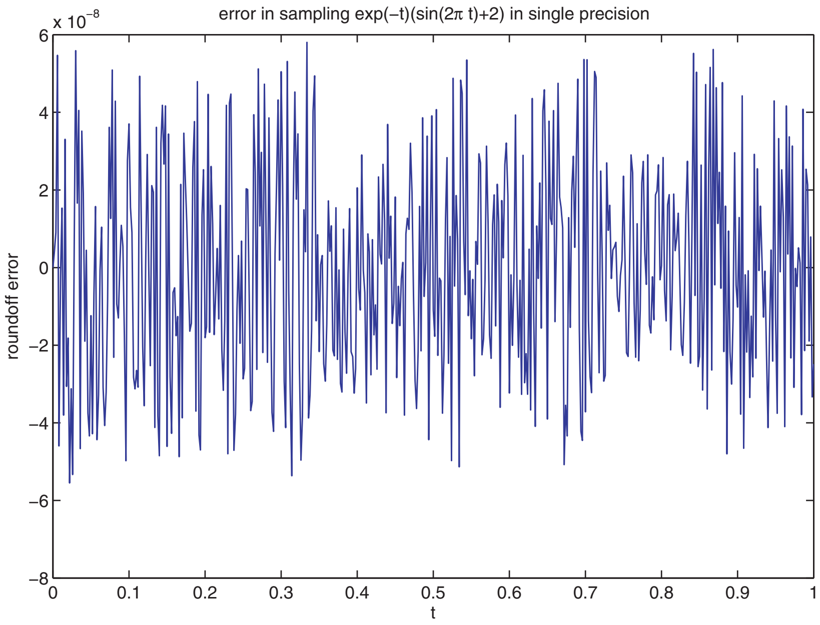

Figure 2.2. *The "almost random" nature of roundoff errors.*

`eps('single')/2`, which yields roughly the value $\eta_{\text{single}} = 6\text{e-}8$. The last line in the script produces approximately the value 0.97, which is indeed close to 1. ■

Specific exercises for this section: Exercises 1–2.

2.2 Floating point systems

Note: Here is where we get more detailed and technical in the present chapter. Note that we are not assuming the IEEE standard before discussing it in Section 2.4.

A floating point system can be characterized by four values (β, t, L, U), where

$$\begin{aligned} \beta &= \text{base of the number system;} \\ t &= \text{precision (\# of digits);} \\ L &= \text{lower bound on exponent } e; \\ U &= \text{upper bound on exponent } e. \end{aligned}$$

Generalizing the binary representation in Section 2.1, we write

$$\text{fl}(x) = \pm\left(\frac{\tilde{d}_0}{\beta^0} + \frac{\tilde{d}_1}{\beta^1} + \cdots + \frac{\tilde{d}_{t-1}}{\beta^{t-1}}\right) \times \beta^e,$$

where β is an integer larger than 1 referred to as the *base* and $\tilde{d}_i$ are integer digits in the range $0 \le \tilde{d}_i \le \beta - 1$. The number fl$(x)$ is an approximation of x. To ensure uniqueness of this representation, it is normalized to satisfy $\tilde{d}_0 \ne 0$ by adjusting the exponent e so that leading zeros are dropped. Note that unless $\beta = 2$ this does not fix the value of $\tilde{d}_0$, which therefore must be stored, too. This is why the last stored digit has index $t-1$ and not t as in Section 2.1. In addition, e must be in the range $L \le e \le U$.

Chopping and rounding

To store $x = \pm(d_0.d_1d_2d_3\cdots d_{t-1}d_td_{t+1}\cdots) \times \beta^e$ using only t digits, it is possible to use one of a number of strategies. The two basic ones are

- *chopping:* ignore digits $d_t, d_{t+1}, d_{t+2}, d_{t+3}\ldots$, yielding $\tilde{d}_i = d_i$ and
$$\text{fl}(x) = \pm\, d_0.d_1d_2d_3\cdots d_{t-1} \times \beta^e;$$
- *rounding:* consult d_t to determine the approximation
$$\text{fl}(x) = \begin{cases} \pm\, d_0.d_1d_2d_3\cdots d_{t-1} \times \beta^e, & d_t < \beta/2, \\ \pm\left(d_0.d_1d_2d_3\cdots d_{t-1} + \beta^{1-t}\right) \times \beta^e, & d_t > \beta/2. \end{cases}$$
In case of a tie ($d_t = \beta/2$), round to the nearest even number.

Example 2.3. Here are results of chopping and rounding with $\beta = 10$, $t = 3$:

x	Chopped to 3 digits	Rounded to 3 digits
5.672	5.67	5.67
−5.672	−5.67	−5.67
5.677	5.67	5.68
−5.677	−5.67	−5.68
5.692	5.69	5.69
5.695	5.69	5.70

■

Example 2.4. Since humans are used to decimal arithmetic, let us set $\beta = 10$. Consider
$$\frac{8}{3} = 2.6666\ldots = \left(\frac{2}{10^0} + \frac{6}{10^1} + \frac{6}{10^2} + \frac{6}{10^3} + \frac{6}{10^4} + \cdots\right) \times 10^0.$$
This number gives an infinite series in base 10, although the digit series has finite length in base 3.

To represent it using $t = 4$, chopping gives
$$\frac{8}{3} \simeq \left(\frac{2}{10^0} + \frac{6}{10^1} + \frac{6}{10^2} + \frac{6}{10^3}\right) \times 10^0 = 2.666 \times 10^0.$$

On the other hand, with rounding note that $d_t = 6 > 5$, so $\beta^{1-t} = 10^{-3}$ is added to the number before chopping, which gives 2.667×10^0.

This floating point representation is not unique; for instance, we have
$$2.666 \times 10^0 = 0.2666 \times 10^1.$$
Therefore, we **normalize** the representation by insisting that $d_0 \neq 0$, so
$$1 \le d_0 \le 9, \;\; 0 \le d_i \le 9, \quad i = 1, \ldots, t-1,$$
which eliminates any ambiguity. ■

The number 0 cannot be represented in a normalized fashion. It and the limits $\pm\infty$ are represented as special combinations of bits, according to an agreed upon convention for the given floating point system.

Example 2.5. Consider a (toy) decimal floating point system with $t = 4$, $U = 1$, and $L = -2$. Thus, the decimal number 2.666 is precisely representable because it has four digits in its mantissa and $L < e < U$.

The largest number here is $99.99 \lessapprox 10^2 = 100$, the smallest is $-99.99 \gtrapprox -100$, and the smallest positive number is $10^{-2} = 0.01$.

How many different numbers do we have? The first digit can take on 9 different values, the other three digits 10 values each (because they may be zero, too). Thus, there are $9 \times 10 \times 10 \times 10 =$ 9,000 different normalized fractions possible. The exponent can be one of $U - L + 1 = 4$ values, so in total there are $4 \times 9{,}000 = 36{,}000$ different positive numbers possible. There is the same total of negative numbers, and then there is the number 0. So, there are 72,001 different numbers in this floating point system. ∎

What about the choice of a base? Computer storage is in an integer multiple of bits, hence all computer representations we know of have used bases that are powers of 2. In the 1970s there were architectures with bases 16 and 8. Today, as we have observed in Section 2.1, the standard floating point system uses base $\beta = 2$; see Figure 2.1 and Section 2.4.

The error in floating point representation

The relative error is generally a more meaningful measure than absolute error in floating point representation, because it is independent of a change of exponent. Thus, if in a decimal system (i.e., with $\beta = 10$) we have that $u = 1{,}000{,}000 = 1 \times 10^6$ and $v = \mathrm{fl}(u) = 990{,}000 = 9.9 \times 10^5$, we expect to be able to work with such an approximation as accurately as with $u = 10$ and $v = \mathrm{fl}(u) = 9.9$. This is borne out by the value of the relative error.

Let us denote the floating point representation mapping by $x \mapsto \mathrm{fl}(x)$, and suppose rounding is used. Then the quantity η in the formula on the current page is fundamental as it expresses a bound on the relative error when we represent a number in our prototype floating point system. We have already mentioned in Section 2.1 the *rounding unit* η. Furthermore, the negative of its exponent, $t - 1$ (for the rounding case), is often referred to as the **number of significant digits**.

Rounding unit.
For a general floating point system (β, t, L, U) the rounding unit is

$$\eta = \frac{1}{2}\beta^{1-t}.$$

Recall that for the double precision word in the standard floating point system, a bound on the relative error is given by $|x - \mathrm{fl}(x)|/|x| \le \eta \approx$ 1.1e-16.

Floating point arithmetic

As briefly mentioned in Section 2.1, the IEEE standard requires **exact rounding**, which means that the result of a basic arithmetic operation must be identical to the result obtained if the arithmetic operation was computed exactly and then the result rounded to the nearest floating point

Theorem: Floating Point Representation Error.
Let $x \mapsto \mathrm{fl}(x) = g \times \beta^e$, where $x \neq 0$ and g is the normalized, signed mantissa.

Then the absolute error committed in using the floating point representation of x is bounded by

$$|x - \mathrm{fl}(x)| \le \begin{cases} \beta^{1-t} \cdot \beta^e & \text{for chopping,} \\ \frac{1}{2}\beta^{1-t} \cdot \beta^e & \text{for rounding,} \end{cases}$$

whereas the relative error satisfies

$$\frac{|x - \mathrm{fl}(x)|}{|x|} \le \begin{cases} \beta^{1-t} & \text{for chopping,} \\ \frac{1}{2}\beta^{1-t} & \text{for rounding.} \end{cases}$$

number. With exact rounding, if $\mathrm{fl}(x)$ and $\mathrm{fl}(y)$ are machine numbers, then

$$\begin{aligned} \mathrm{fl}(\mathrm{fl}(x) \pm \mathrm{fl}(y)) &= (\mathrm{fl}(x) \pm \mathrm{fl}(y))(1+\epsilon_1), \\ \mathrm{fl}(\mathrm{fl}(x) \times \mathrm{fl}(y)) &= (\mathrm{fl}(x) \times \mathrm{fl}(y))(1+\epsilon_2), \\ \mathrm{fl}(\mathrm{fl}(x) \div \mathrm{fl}(y)) &= (\mathrm{fl}(x) \div \mathrm{fl}(y))(1+\epsilon_3), \end{aligned}$$

where $|\epsilon_i| \le \eta$. Thus, the *relative errors* remain small after each such operation.

Example 2.6. Consider a floating point system with decimal base $\beta = 10$ and four digits, i.e., $t = 4$. Thus, the rounding unit is $\eta = \frac{1}{2} \times 10^{-3}$. Let

$$x = .1103 = 1.103 \times 10^{-1}, \quad y = 9.963 \times 10^{-3}.$$

Subtracting these two numbers, the exact value is $x - y = .100337$. Hence, exact rounding yields .1003. Obviously, the relative error is

$$\frac{|.100337 - .1003|}{.100337} \approx .37 \times 10^{-3} < \eta.$$

However, if we were to subtract these two numbers without guard digits we would obtain $.1103 - .0099 = .1004$. Now the obtained relative error is not below η, because

$$\frac{|.100337 - .1004|}{.100337} \approx .63 \times 10^{-3} > \eta.$$

Thus, guard digits must be used to produce exact rounding. ∎

Example 2.7. Generally, proper rounding yields $\mathrm{fl}(1+\alpha) = 1$ for any number α that satisfies $|\alpha| \le \eta$. In particular, the MATLAB commands

```
eta = .5*2^(-52), beta = (1+eta)-1
```

produce the output `eta` = 1.1102e-16, `beta` = 0. Returning to Example 1.3 and to Figure 1.3, we can now explain why the curve of the error is flat for the very, very small values of h. For such values, $\mathrm{fl}(f(x_0+h)) = \mathrm{fl}(f(x_0))$, so the approximation is precisely zero and the recorded values are those of $\mathrm{fl}(f'(x_0))$, which is independent of h. ∎

Spacing of floating point numbers

If you think of how a given floating point system represents the real line you'll find that it has a somewhat uneven nature. Indeed, very large numbers are not represented at all, and the distance between two neighboring, positive floating point values is constant in the relative but not in the absolute sense.

Example 2.8. Let us plot the decimal representation of the numbers in the system specified by $(\beta, t, L, U) = (2, 3, -2, 3)$. The smallest possible number in the mantissa $d_0.d_1d_2$ is 1.00 and the largest possible number is 1.11 in binary, which is equal to $1 + 1/2 + 1/4 = 1.75$. So, we will run in a loop from 1 to 1.75 in increments of $\beta^{1-t} = 2^{-2} = 0.25$. This is the basis for one loop. The exponent runs from -2 to 3, which is what we loop over in the second (nested) loop. The resulting double loop builds up an array with all the positive elements of the system.

Here is a MATLAB script that plots the numbers of the system. Notice the format in the `plot` command.

```
x = [];

% Generate all positive numbers of the system (2,3,-2,3)
for i = 1:0.25:1.75
  for j = -2:3
      x = [x i*2^j];
  end
end

x=[x -x 0];      % Add all negative numbers and 0
x = sort(x);     % Sort
y = zeros(1,length(x));
plot(x,y,'+')
```

The resulting plot is in Figure 2.3. Note that the points are not distributed uniformly along the real line.

The rounding unit for this system is

$$\eta = \frac{1}{2}\beta^{1-t} = \frac{1}{2} \times 2^{1-3} = \frac{1}{8},$$

which is half the spacing between $1 = \beta^0$ and the next number in the system. (Can you see why?) ∎

Most annoying in Figure 2.3 is the fact that the distance between the value 0 and the smallest positive number is significantly *larger* than the distance between that same smallest positive number and the next smallest positive number! A commonly used way to remedy this in modern floating point systems is by introducing next to 0 additional, *subnormal numbers* which are not normalized. We will leave it at that.

MATLAB has a parameter called `eps` that signifies the spacing of floating point numbers. In particular, `eps`(1) gives twice the rounding unit η. Enter `help eps` to learn more about this parameter.

Specific exercises for this section: Exercises 3–8.

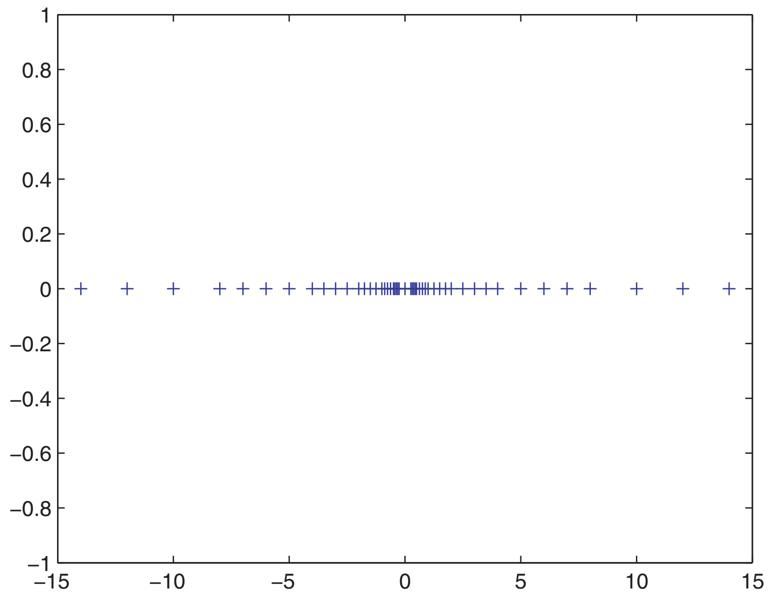

Figure 2.3. *Picture of the floating point system described in Example* 2.8.

2.3 Roundoff error accumulation

Because many operations are being performed in the course of carrying out a numerical algorithm, many small errors unavoidably result. We know that each elementary floating point operation may introduce a small relative error, but how do these errors accumulate?

In general, as we have already mentioned in Chapter 1, error growth that is linear in the number of operations is unavoidable. Yet, there are a few things to watch out for.

- Error magnification:

  1. If x and y differ widely in magnitude, then $x + y$ has a large absolute error.
  2. If $|y| \ll 1$, then x/y may have large relative and absolute errors. Likewise for xy if $|y| \gg 1$.
  3. If $x \simeq y$, then $x - y$ has a large relative error (*cancellation error*).

- Overflow and underflow:

  An **overflow** is obtained when a number is too large to fit into the floating point system in use, i.e., when $e > U$. An **underflow** is obtained when $e < L$. When overflow occurs in the course of a calculation, this is generally fatal. But underflow is nonfatal: the system usually sets the number to 0 and continues. (MATLAB does this, quietly.)

It is also worth mentioning again the unfortunate possibility of running into `NaN`, described in Section 2.1.

Design of library functions

You may wonder what the fuss is all about, if the representation of a number and basic arithmetic operations cause an error as small as about 10^{-16} in a typical floating point system. But taking this issue lightly may occasionally cause surprisingly serious damage. In some cases, even if the computation is long and intensive it may be based on or repeatedly use a short algorithm, and a simple change in a formula may greatly improve performance with respect to roundoff error accumulation. For those operations that are carried out repeatedly in a typical calculation, it may well be worth obtaining a result as accurate as possible, even at the cost of a small computational overhead.

Example 2.9. The most common way to measure the size of a vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)^T$ is the Euclidean norm (also called ℓ_2-norm) defined by

$$\|\mathbf{x}\|_2 = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}.$$

We discuss vector norms and their properties in Section 4.2. Any standard numerical linear algebra software package has a library function that computes the norm, and often the input vector is very large; see, for instance, Chapter 7 for relevant situations. One of the dangers to avoid in computing the norm is the possibility of overflow. To illustrate this point, let us look at a small vector with just two but widely differing components.

Consider computing $c = \sqrt{a^2 + b^2}$ in a floating point system with four decimal digits and two exponent digits, for $a = 10^{60}$ and $b = 1$. Thus, c is the Euclidean norm of the vector $\mathbf{x} = (10^{60}, 1)^T$. The correct result here is $c = 10^{60}$. But overflow will result during the course of calculation, because squaring a gives 10^{120}, which cannot be represented in this system. Yet, this overflow can easily be avoided if we rescale ahead of time, noting that $c = s\sqrt{(a/s)^2 + (b/s)^2}$ for any $s \neq 0$. Thus, using $s = a = 10^{60}$ gives an underflow when b/s is squared, which is set to zero. This yields the correct answer. (It is important to stress that "correct answer" here refers not necessarily to the "true," precise answer on the real line, but rather to the most accurate answer we can obtain given this particular floating point system.)

The above principle can be generalized to the computation of the norm of a vector of more than two components, simply scaling by the largest one.

Another issue is the order in which the values x_i^2 are summed. Doing this in increasing order of magnitude reduces error accumulation; see, for instance, Exercise 17.

In Exercise 18 you are asked to design a library function for computing the ℓ_2-norm of a vector. ∎

Avoiding cancellation error

Another trouble spot to watch out for is cancellation error, mentioned in Section 2.1. Some insight into the damaging effect of subtracting two nearly equal numbers in a floating point system can be observed by deriving a bound on the error. Suppose $z = x - y$, where $x \approx y$. Then

$$|z - \mathrm{fl}(z)| \leq |x - \mathrm{fl}(x)| + |y - \mathrm{fl}(y)|\,,$$

from which it follows that the relative error satisfies

$$\frac{|z - \mathrm{fl}(z)|}{|z|} \leq \frac{|x - \mathrm{fl}(x)| + |y - \mathrm{fl}(y)|}{|x - y|}\,.$$

The numerator of the bound could be tight since it merely depends on how well the floating point system can represent the numbers x and y. But the denominator is very close to zero if $x \approx y$ regardless of how well the floating point system can work for these numbers, and so the relative error in z could become large.

Example 2.10. The roots of the quadratic equation

$$x^2 - 2bx + c = 0$$

with $b^2 > c$ are given by

$$x_{1,2} = b \pm \sqrt{b^2 - c}.$$

The following simple script gives us the roots according to the formula:

```
x1 = b + sqrt(b^2-c);
x2 = b - sqrt(b^2-c);
```

Unfortunately, if b^2 is significantly larger than c, to the point that $\sqrt{b^2-c} \approx |b|$, then one of the two calculated roots is bound to be approximately zero and inaccurate. (There is no problem with the other root, which would be approximately equal to $2b$ in this case.)

To our aid comes the formula $x_1x_2 = c$. We can then avoid the cancellation error by using a modified algorithm, given by the script

```
if b > 0
  x1 = b + sqrt(b^2-c);
  x2 = c / x1;
else
  x2 = b - sqrt(b^2-c);
  x1 = c / x2;
end
```

The algorithm can be further improved by also taking into consideration the possibility of an overflow. Thus, if $b^2 \gg c$, we replace $\sqrt{b^2-c}$ by $|b|\sqrt{1-\frac{c}{b^2}}$, in a way similar to the technique described in Example 2.9. In Exercise 16 you are asked to design and implement a robust quadratic equation solver. ■

Cancellation error deserves special attention because it often appears in practice in an identifiable way. For instance, recall Example 1.2. If we approximate a derivative of a smooth (differentiable) function $f(x)$ at a point $x = x_0$ by the difference of two neighboring values of f divided by the difference of the arguments with a small step h, e.g., by

$$f'(x_0) \approx \frac{f(x_0+h) - f(x_0)}{h},$$

then there is a cancellation error in the numerator, which is then magnified by the denominator. Recall also the discussion in Section 2.1, just before Example 2.2.

When cancellation errors appear in an identifiable way, they can often be avoided by a simple modification in the algorithm. An instance is illustrated in Exercise 2. Here is another one.

Example 2.11. Suppose we wish to compute $y = \sqrt{x+1} - \sqrt{x}$ for $x = 100{,}000$ in a five-digit decimal arithmetic. Clearly, the number 100,001 cannot be represented in this floating point system exactly, and its representation in the system (when either chopping or rounding is used) is 100,000. In other words, for this value of x in this floating point system, we have $x + 1 = x$. Thus, naively computing $\sqrt{x+1} - \sqrt{x}$ results in the value 0.

We can do much better if we use the identity

$$\frac{(\sqrt{x+1} - \sqrt{x})(\sqrt{x+1} + \sqrt{x})}{(\sqrt{x+1} + \sqrt{x})} = \frac{1}{\sqrt{x+1} + \sqrt{x}}.$$

Applying this formula (i.e., computing the right-hand expression in 5-digit decimal arithmetic) yields 1.5811×10^{-3}, which happens to be the correct value to 5 decimal digits. ∎

There are also other ways to avoid cancellation error, and one popular technique is the use of a Taylor expansion (page 5).

Example 2.12. Suppose we wish to design a library function for computing

$$y = \sinh(x) = \frac{1}{2}(e^x - e^{-x}).$$

Think, for example, of having a scientific calculator: we want to be sure that it gives an accurate result for any argument x.

While the above formula can be straightforwardly applied (assuming we have a decent library function for computing exponents), the going may get a little tougher for values of x near 0. Directly using the formula for computing y may be prone to severe cancellation errors, due to the subtraction of two quantities that are approximately equal to 1. On the other hand, using the Taylor expansion

$$\sinh(x) = x + \frac{x^3}{6} + \frac{\xi^5}{120}$$

for some ξ satisfying $|\xi| \le |x|$ may prove useful. The simple cubic expression $x + \frac{x^3}{6}$ should give an effective approximation if $|x|$ is sufficiently small, as the discretization error can be bounded by $\frac{|x|^5}{120}$ and the roundoff error is no longer an issue.

Employing 5-digit decimal arithmetic for our cubic approximation, we compute $\sinh(0.1) = 0.10017$ and $\sinh(0.01) = 0.01$. (Believe us! Or otherwise please feel free to verify.) These are the "exact" values in this floating point system. On the other hand, using the formula that involves the exponential functions, which is the *exact* formula and would produce the exact result had we not had roundoff errors, we obtain 0.10018 and 0.010025 for these two values of x, respectively, so there are pronounced errors.

For $x = 0.01$ it is in fact sufficient to use only one term of the Taylor expansion, i.e., approximate $\sinh(x)$ by x in this toy floating point system. ∎

Specific exercises for this section: Exercises 9–19.

2.4 The IEEE standard

During the dawn of the modern computing era there was chaos. Then, in 1985, came the IEEE standard, slightly updated in 2008. Today, except for special circumstances such as a calculator or a dedicated computer on a special-purpose device, every software designer and hardware manufacturer follows this standard. Thus, the output of one's program no longer varies upon switching laptops or computing environments using the same language. Here we describe several of the nuts and bolts (from among the less greasy ones) of this standard floating point system.

Recall from Section 2.1 that the base is $\beta = 2$. In decimal value we have

$$\text{fl}(x) = \pm\left(1 + \frac{\tilde{d}_1}{2} + \frac{\tilde{d}_2}{4} + \cdots + \frac{\tilde{d}_t}{2^t}\right) \times 2^e,$$

where $\tilde{d}_i$ are binary digits each requiring one bit for storing.

The standard and the single precision floating point words

The standard floating point word used exclusively for all calculations in this book except Examples 2.2 and 14.7 requires 64 bits of storage and is called *double precision* or *long word*. This is the MATLAB default. Of these 64 bits, one is allocated for sign s (the number is negative if and only if $s = 1$), 11 for the exponent, and $t = 52$ for the fraction:

Double precision (64-bit word)

$s = \pm$	$b = 11$-bit exponent	$f = 52$-bit fraction

$\beta = 2$, $t = 52$, $L = -1022$, $U = 1023$

Since the fraction f contains t digits the precision is $52+1 = 53$. Thus, a given bit pattern in the last 52 bits of a long word is interpreted as a sequence of binary digits $d_1 d_2 \cdots d_{52}$. A given bit pattern in the exponent part of the long word is interpreted as a positive integer b in binary representation that yields the exponent e upon shifting by U, i.e., $e = b - 1023$.

Example 2.13. The sequence of 64 binary digits

$$0100000001111110100000000000000000000000000000000000000000000000$$

considered as a double precision word can be interpreted as follows:

- Based on the first digit 0 the sign is positive by convention.
- The next 11 digits,
  $$10000000111,$$
  form the exponent: in decimal, $b = 1 \times 2^{10} + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 1031$. So in decimal, $e = b - 1023 = 1031 - 1023 = 8$.
- The next 52 bits,
  $$1110100000000000000000000000000000000000000000000000,$$
  form the decimal fraction: $f = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{32} = 0.90625$.
- The number in decimal is therefore
  $$s \cdot (1+f) \times 2^e = 1.90625 \times 2^8 = 488. \quad \blacksquare$$

There is also a *single precision* arrangement of 32 bits, as follows:

Single precision (32-bit word)

$s = \pm$	$b = 8$-bit exponent	$f = 23$-bit fraction

$\beta = 2$, $t = 23$, $L = -126$, $U = 127$

Storing special values

Upon carefully examining the possible range of exponents you will notice that it is not fully utilized. For double precision $2^{11} = 2048$ different exponents could be distinguished, and for single precision $2^8 = 256$ are available. But only 2046 and 254, respectively, are used in practice. This is because the IEEE standard reserves the endpoints of the exponent range for special purposes. The largest number precisely representable in a standard long word is therefore

$$\left[1+\sum_{i=1}^{52}\left(\frac{1}{2}\right)^i\right] \times 2^{1023} = (2-2^{-52}) \times 2^{1023} \approx 2^{1024} \approx 10^{308},$$

and the smallest positive number is

$$[1+0] \times 2^{-1022} = 2^{-1022} \approx 2.2 \times 10^{-308}.$$

How are 0 and ∞ stored? Here is where the endpoints of the exponent are used, and the conventions are simple and straightforward:

- For 0, set $b = 0, f = 0$, with s arbitrary; i.e., the minimal positive value representable in the system is considered 0.
- For $\pm\infty$, set $b = 1\cdots 1, f = 0$.
- The pattern $b = 1\cdots 1,\ f \neq 0$ is by convention `NaN`.

In single precision, or short word, the largest number precisely representable is

$$\left[1+\sum_{i=1}^{23}\left(\frac{1}{2}\right)^i\right] \times 2^{127} = (2-2^{-23}) \times 2^{127} \approx 2^{128} \approx 3.4 \times 10^{38},$$

and the smallest positive number is

$$[1+0] \times 2^{-126} = 2^{-126} \approx 1.2 \times 10^{-38}.$$

Rounding unit

The formulas for the machine precision or rounding unit η were introduced on page 19 in Section 2.1. Note again the shift from $t-1$ to t in the power of the base as compared to the general formula given on page 23.

Using the word arrangement for single and double precision, η is therefore calculated as follows:

- For single precision, $\eta = \frac{1}{2} \cdot \beta^{-t} = \frac{1}{2} \cdot 2^{-23} \approx 6.0 \times 10^{-8}$ (so, there are 23 significant binary digits, or about 7 decimal digits).
- For double precision, $\eta = \frac{1}{2} \cdot \beta^{-t} = \frac{1}{2} \cdot 2^{-52} \approx 1.1 \times 10^{-16}$ (so, there are 52 significant binary digits, or about 16 decimal digits).

Typically, single and double precision floating point systems as described above are implemented in hardware. There is also quadruple precision (128 bits), often implemented in software and thus considerably slower, for applications that require very high precision (e.g., in semiconductor simulation, numerical relativity and astronomical calculations).

The fundamentally important *exact rounding*, mentioned in both Sections 2.1 and 2.2, has a rather lengthy definition for its implementation, which stands in contrast to the cleanly stated requirement of its result. We will not dive deeper into this.

Specific exercises for this section: Exercises 20–21.

2.5 Exercises

0. **Review questions**

   (a) What is a normalized floating point number and what is the purpose of normalization?

   (b) A general floating point system is characterized by four values (β, t, L, U). Explain in a few brief sentences the meaning and importance of each of these parameters.

   (c) Write down the floating point representation of a given real number x in a decimal system with $t = 4$, using (i) chopping and (ii) rounding.

   (d) Define rounding unit (or machine precision) and explain its importance.

   (e) Define overflow and underflow. Why is the former considered more damaging than the latter?

   (f) What is a cancellation error? Give an example of an application where it arises in a natural way.

   (g) What is the rounding unit for base $\beta = 2$ and $t = 52$ digits?

   (h) Under what circumstances could nonnormalized floating point numbers be desirable?

   (i) Explain the storage scheme for single precision and double precision numbers in the IEEE standard.

1. The fraction in a single precision word has 23 bits (alas, *less* than half the length of the double precision word).

   Show that the corresponding rounding unit is approximately 6×10^{-8}.

2. The function $f_1(x_0, h) = \sin(x_0 + h) - \sin(x_0)$ can be transformed into another form, $f_2(x_0, h)$, using the trigonometric formula

$$\sin(\phi) - \sin(\psi) = 2\cos\left(\frac{\phi + \psi}{2}\right)\sin\left(\frac{\phi - \psi}{2}\right).$$

   Thus, f_1 and f_2 have the same values, in exact arithmetic, for any given argument values x_0 and h.

   (a) Derive $f_2(x_0, h)$.

   (b) Suggest a formula that avoids cancellation errors for computing the approximation $(f(x_0 + h) - f(x_0))/h$ to the derivative of $f(x) = \sin(x)$ at $x = x_0$. Write a MATLAB program that implements your formula and computes an approximation of $f'(1.2)$, for $h = $ 1e-20,1e-19,…,1.

   (c) Explain the difference in accuracy between your results and the results reported in Example 1.3.

3. (a) How many distinct positive numbers can be represented in a floating point system using base $\beta = 10$, precision $t = 2$ and exponent range $L = -9$, $U = 10$?

   (Assume normalized fractions and don't worry about underflow.)

   (b) How many normalized numbers are represented by the floating point system (β, t, L, U)? Provide a formula in terms of these parameters.

4. Suppose a computer company is developing a new floating point system for use with their machines. They need your help in answering a few questions regarding their system. Following the terminology of Section 2.2, the company's floating point system is specified by (β, t, L, U). Assume the following:

- All floating point values are normalized (except the floating point representation of zero).
- All digits in the mantissa (i.e., fraction) of a floating point value are explicitly stored.
- The number 0 is represented by a float with a mantissa and an exponent of zeros. (Don't worry about special bit patterns for $\pm\infty$ and `NaN`.)

Here is your part:

(a) How many different nonnegative floating point values can be represented by this floating point system?

(b) Same question for the actual choice $(\beta, t, L, U) = (8, 5, -100, 100)$ (in decimal) which the company is contemplating in particular.

(c) What is the approximate value (in decimal) of the largest and smallest positive numbers that can be represented by this floating point system?

(d) What is the rounding unit?

5. (a) The number $\frac{8}{7} = 1.14285714285714\ldots$ obviously has no exact representation in any decimal floating point system ($\beta = 10$) with finite precision t. Is there a finite floating point system (i.e., some finite integer base β and precision t) in which this number does have an exact representation? If yes, then describe such a system.

   (b) Answer the same question for the irrational number π.

6. Write a MATLAB program that receives as input a number x and a parameter n and returns x rounded to n decimal digits. Write your program so that it can handle an array as input, returning an array of the same size in this case.

   Use your program to generate numbers for Example 2.2, demonstrating the phenomenon depicted there without use of single precision.

7. Prove the Floating Point Representation Error Theorem on page 24.

8. Rewrite the script of Example 2.8 without any use of loops, using vectorized operations instead.

9. Suggest a way to determine approximately the rounding unit of your calculator. State the type of calculator you have and the rounding unit you have come up with. If you do not have a calculator, write a short MATLAB script to show that your algorithm works well on the standard IEEE floating point system.

10. The function $f_1(x, \delta) = \cos(x + \delta) - \cos(x)$ can be transformed into another form, $f_2(x, \delta)$, using the trigonometric formula

$$\cos(\phi) - \cos(\psi) = -2\sin\left(\frac{\phi + \psi}{2}\right)\sin\left(\frac{\phi - \psi}{2}\right).$$

   Thus, f_1 and f_2 have the same values, in exact arithmetic, for any given argument values x and δ.

   (a) Show that, analytically, $f_1(x, \delta)/\delta$ or $f_2(x, \delta)/\delta$ are effective approximations of the function $-\sin(x)$ for δ sufficiently small.

   (b) Derive $f_2(x, \delta)$.

   (c) Write a MATLAB script which will calculate $g_1(x, \delta) = f_1(x, \delta)/\delta + \sin(x)$ and $g_2(x, \delta) = f_2(x, \delta)/\delta + \sin(x)$ for $x = 3$ and $\delta = 1.\text{e-}11$.

   (d) Explain the difference in the results of the two calculations.

11. (a) Show that
$$\ln\left(x-\sqrt{x^2-1}\right) = -\ln\left(x+\sqrt{x^2-1}\right) .$$

    (b) Which of the two formulas is more suitable for numerical computation? Explain why, and provide a numerical example in which the difference in accuracy is evident.

12. For the following expressions, state the numerical difficulties that may occur, and rewrite the formulas in a way that is more suitable for numerical computation:

    (a) $\sqrt{x+\frac{1}{x}} - \sqrt{x-\frac{1}{x}}$, where $x \gg 1$.

    (b) $\sqrt{\frac{1}{a^2}+\frac{1}{b^2}}$, where $a \approx 0$ and $b \approx 1$.

13. Consider the linear system
$$\begin{pmatrix} a & b \\ b & a \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$
with $a,b > 0$; $a \neq b$.

    (a) If $a \approx b$, what is the numerical difficulty in solving this linear system?

    (b) Suggest a numerically stable formula for computing $z = x+y$ given a and b.

    (c) Determine whether the following statement is true or false, and explain why:

    "When $a \approx b$, the problem of solving the linear system is ill-conditioned but the problem of computing $x+y$ is not ill-conditioned."

14. Consider the approximation to the first derivative
$$f'(x) \approx \frac{f(x+h)-f(x)}{h}.$$
The *truncation* (or *discretization*) error for this formula is $\mathcal{O}(h)$. Suppose that the absolute error in evaluating the function f is bounded by ε and let us ignore the errors generated in basic arithmetic operations.

    (a) Show that the total computational error (truncation and rounding combined) is bounded by
$$\frac{Mh}{2}+\frac{2\varepsilon}{h},$$
    where M is a bound on $|f''(x)|$.

    (b) What is the value of h for which the above bound is minimized?

    (c) The rounding unit we employ is approximately equal to 10^{-16}. Use this to explain the behavior of the graph in Example 1.3. Make sure to explain the shape of the graph as well as the value where the apparent minimum is attained.

(d) It is not difficult to show, using Taylor expansions, that $f'(x)$ can be approximated more accurately (in terms of truncation error) by

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}.$$

For this approximation, the truncation error is $\mathcal{O}(h^2)$. Generate a graph similar to Figure 1.3 (please generate only the solid line) for the same function and the same value of x, namely, for sin(1.2), and compare the two graphs. Explain the meaning of your results.

15. Suppose a machine with a floating point system $(\beta, t, L, U) = (10, 8, -50, 50)$ is used to calculate the roots of the quadratic equation

$$ax^2 + bx + c = 0,$$

where a, b, and c are given, real coefficients.

For each of the following, state the numerical difficulties that arise if one uses the standard formula for computing the roots. Explain how to overcome these difficulties (when possible).

(a) $a = 1$; $b = -10^5$; $c = 1$.

(b) $a = 6 \cdot 10^{30}$; $b = 5 \cdot 10^{30}$; $c = -4 \cdot 10^{30}$.

(c) $a = 10^{-30}$; $b = -10^{30}$; $c = 10^{30}$.

16. Write a quadratic equation solver. Your MATLAB script should get a,b,c as input, and accurately compute the roots of the corresponding quadratic equation. Make sure to check end cases such as $a = 0$, and consider ways to avoid an overflow and cancellation errors. Implement your algorithm and demonstrate its performance on a few cases (for example, the cases mentioned in Exercise 15). Show that your algorithm produces better results than the standard formula for computing roots of a quadratic equation.

17. Write a MATLAB program that

(a) sums up $1/n$ for $n = 1, 2, \ldots, 10{,}000$;

(b) rounds each number $1/n$ to 5 decimal digits and then sums them up *in 5-digit decimal arithmetic* for $n = 1, 2, \ldots, 10{,}000$;

(c) sums up the same rounded numbers (in 5-digit decimal arithmetic) in reverse order, i.e., for $n = 10{,}000, \ldots, 2, 1$.

Compare the three results and explain your observations. For generating numbers with the requested precision, you may want to do Exercise 6 first.

18. (a) Explain in detail how to avoid overflow when computing the ℓ_2-norm of a (possibly large in size) vector.

(b) Write a MATLAB script for computing the norm of a vector in a numerically stable fashion. Demonstrate the performance of your code on a few examples.

19. In the statistical treatment of data one often needs to compute the quantities

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i, \quad s^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2,$$

where $x_1, x_2, \ldots, x_n$ are the given data. Assume that n is large, say, $n = 10{,}000$. It is easy to see that s^2 can also be written as

$$s^2 = \frac{1}{n}\sum_{i=1}^{n} x_i^2 \; - \; \bar{x}^2.$$

(a) Which of the two methods to calculate s^2 is cheaper in terms of overall computational cost? Assume $\bar{x}$ has already been calculated and give the operation counts for these two options.

(b) Which of the two methods is expected to give more accurate results for s^2 in general?

(c) Give a small example, using a decimal system with precision $t = 2$ and numbers of your choice, to validate your claims.

20. With exact rounding, we know that each *elementary* operation has a relative error which is bounded in terms of the rounding unit η; e.g., for two floating point numbers x and y, $\mathrm{fl}(x + y) = (x + y)(1 + \epsilon)$, $|\epsilon| \le \eta$. But is this true also for elementary functions such as sin, ln, and exponentiation?

    Consider exponentiation, which is performed according to the formula

    $$x^y = e^{y \ln x} \quad \text{(assuming } x > 0\text{)}.$$

    Estimate the relative error in calculating x^y in floating point, assuming $\mathrm{fl}(\ln z) = (\ln z)(1 + \epsilon)$, $|\epsilon| \le \eta$, and that everything else is exact. Show that the sort of bound we have for elementary operations and for ln does not hold for exponentiation when x^y is very large.

21. The IEEE 754 (known as the floating point standard) specifies the 128-bit word as having 15 bits for the exponent.

    What is the length of the fraction? What is the rounding unit? How many significant decimal digits does this word have?

    Why is quadruple precision more than twice as accurate as double precision, which is in turn more than twice as accurate as single precision?

2.6 Additional notes

A lot of thinking in the early days of modern computing went into the design of floating point systems for computers and scientific calculators. Such systems should be economical (fast execution in hardware) on one hand, yet they should also be reliable, accurate enough, and free of unusual exception-handling conventions on the other hand. W. Kahan was particularly instrumental in such efforts (and received a Turing award for his contributions), especially in setting up the IEEE standard. The almost universal adoption of this standard has significantly increased both reliability and portability of numerical codes. See Kahan's webpage for various interesting related documents: http://www.cs.berkeley.edu/$\sim$wkahan/.

A short, accessible textbook that discusses floating point systems in great detail is Overton [58]. A comprehensive and thorough treatment of roundoff errors and many aspects of numerical stability can be found in Higham [40].

The practical way of working with floating point arithmetic, which is to attempt to keep errors "small enough" so as not be a bother, is hardly satisfactory from a theoretical point of view. Indeed, what if we want to use a floating point calculation for the purpose of producing a mathematical proof?! The nature of the latter is that a stated result should always—not just usually—hold true. A more careful approach uses **interval arithmetic**. With each number are associated a lower and an upper bound, and these are propagated with the algorithm calculations on a “worst case scenario” basis. The calculated results are then guaranteed to be within the limits of the calculated bounds. See Moore, Kearfott, and Cloud [54] and also [58] for an introduction to this veteran subject. Naturally, such an approach is expensive and can be of limited utility, as the range between the bounds typically grows way faster than the accumulated error itself. But at times this approach does work for obtaining truly guaranteed results.

A move to put (more generally) **complexity theory** for numerical algorithms on firmer foundations was initiated by S. Smale and others in the 1980s; see [9]. These efforts have mushroomed into an entire organization called Foundations of Computational Mathematics (FOCM) that is involved with various interesting activities which concentrate on the more mathematically rich aspects of numerical computation.