Chapter 1

Numerical Algorithms and Roundoff Errors

In this chapter we introduce and discuss some basic concepts of scientific computing. We begin with a general, brief introduction to the field and how it is related to other scientific disciplines. We then get into a detailed discussion of the most fundamental source of imperfection in numerical computing: roundoff errors.

1.1 Numerical analysis and the art of scientific computing

Scientific computing is a discipline concerned with the development and study of **numerical algorithms** for solving mathematical problems that arise in various disciplines in science and engineering.

Typically, the starting point is a given **mathematical model** which has been formulated in an attempt to explain and understand an *observed phenomenon* in biology, chemistry, physics, economics, or any engineering or scientific discipline. We will concentrate on those mathematical models which are *continuous* (or *piecewise continuous*) and are difficult or impossible to solve analytically: this is usually the case in practice. Relevant application areas within computer science include graphics, vision and motion analysis, image and signal processing, search engines and data mining, machine learning, hybrid and embedded systems, and more.

In order to solve such a model approximately on a computer, the (continuous, or piecewise continuous) problem is approximated by a discrete one. Continuous functions are approximated by finite arrays of values. Algorithms are then sought which approximately solve the mathematical problem **efficiently, accurately** and **reliably**. While scientific computing focuses on the design and the implementation of such algorithms, **numerical analysis**

may be viewed as the theory behind them.

The next step after devising suitable algorithms is their implementation. This leads to questions involving programming languages, data structures, computing architectures and their exploitation (by suitable algorithms), etc. The big picture is depicted in Figure 1.1.

The set of requirements that good scientific computing algorithms must satisfy, which seems elementary and obvious, may actually pose rather difficult and complex practical challenges. The main purpose of these notes is to equip you with basic methods and analysis tools for handling such challenges as they may arise in future endeavours.

In terms of computing tools, we will be using MATLAB: This is an interactive computer language, which for our purposes may best be viewed as a convenient **problem solving environment**. Its original concept was very simple: only an array data structure is considered. Most standard linear algebra problems, such as solving linear systems that are not very large, are implemented as part of the environment and are available by a single simple command. These days, MATLAB is much more than a language based on simple data arrays; it is truly a complete environment. Its interactivity and graphics capabilities make it much more suitable and convenient in our context than general-purpose languages such as `C++`, `Java` or `Fortran 90`. In fact, many of the algorithms that we will learn are already implemented in MATLAB... *So why learn them at all*?? Because they provide the basis for much more complex tasks, not quite available (that is to say, not already solved) in MATLAB or anywhere else, which you may encounter in the future.

Rather than producing yet another MATLAB Tutorial or Introduction in these notes (there are several very good ones available in other texts as well as on the internet) we will demonstrate the use of this language on examples as we go along.

1.2 Numerical algorithms and errors

The most fundamental feature of numerical computing is the inevitable presence of error. The result of any interesting computation (and of many uninteresting ones) will be only approximate, and our general quest is to ensure that the resulting error be tolerably small. In this section we discuss the following topics:

- Error types
- Ways to measure errors: Relative and absolute errors
- Algorithm properties

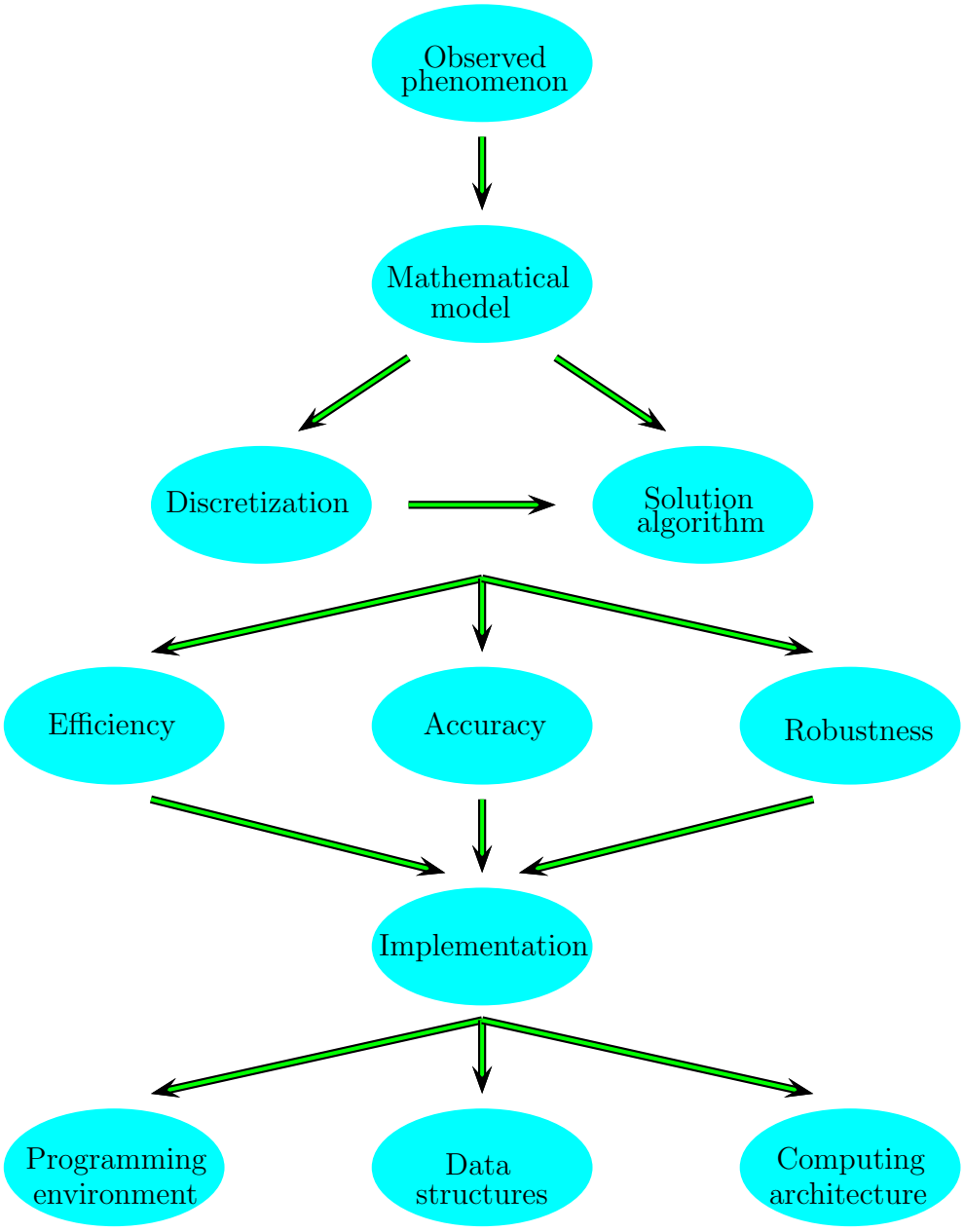

Figure 1.1: Scientific computing.

Error types

There are several types of error that may limit the accuracy of a numerical calculation.

1. **Errors in the problem to be solved**.

   These may be approximation **errors in the mathematical model**. For instance:

   - Heavenly bodies are often approximated by spheres when calculating their properties; an example can be conceived within the approximate calculation of their motion trajectory, attempting to answer the question (say) whether a particular astroid will collide with Planet Earth before 11.12.2011.
   - Relatively unimportant chemical reactions are often discarded in complex chemical modeling in order to arrive at a mathematical problem of a manageable size.

   It is important to realize, then, that often approximation errors of the type stated above are deliberately made: The assumption is that simplification of the problem is worthwhile even if it generates an error in the model. Note, however, that we are still talking about the mathematical model itself; approximation errors related to the numerical solution of the problem are to be discussed below.

   Another typical source of error is **error in the input data**. This may arise, for instance, from physical measurements, which are never infinitely accurate.

   Thus, it may occur that after careful numerical solution of a given problem, the resulting solution would not quite match observations on the phenomenon being examined.

   At the level of numerical algorithms, which is the focus of our interest here, there is really nothing we can do about such errors. However, they should be taken into consideration, for instance when determining the accuracy (tolerance with respect to the next two types of error mentioned below) to which the numerical problem should be solved.

2. **Approximation errors**

   Such errors arise when an approximate formula is used in place of the actual function to be evaluated. There are two types of approximation errors.

   - **Discretization errors** arise from discretizations of continuous processes, such as interpolation, differentiation and integration.

Taylor's Series Theorem: Assume that $f(x)$ has $k+1$ derivatives in an interval containing the points x_0 and $x_0 + h$. Then

$$\begin{aligned} f(x_0+h) &= f(x_0) + hf'(x_0) + \frac{h^2}{2}f''(x_0) + \cdots + \frac{h^k}{k!}f^{(k)}(x_0) + \\ &+ \frac{h^{k+1}}{(k+1)!}f^{(k+1)}(\xi) \end{aligned}$$

where ξ is some point between x_0 and $x_0 + h$.

- **Convergence errors** arise in iterative methods. For instance, nonlinear problems must generally be solved approximately by an iterative process. Such a process would converge to the exact solution in the limit (after infinitely many iterations), but we cut it of course after a finite (hopefully small!) number of iterations. Iterative methods often arise already in *linear* algebra, where an iterative process is terminated after a finite number of iterations before the exact solution is reached.

3. **Roundoff errors**

   Any computation involving real, as opposed to integer, numbers, involves roundoff error. Even when no approximation error is involved (as in the direct evaluation of a straight line, or the solution by Gaussian elimination of a linear system of equations), roundoff errors are present. These arise because of the finite precision representation of real numbers on any computer, which affects both data representation and computer arithmetic. They are further discussed in Section 1.3.

Discretization and convergence errors may be assessed by analysis of the method used, and we will see a lot of that. Unlike roundoff errors, they have a relatively smooth structure which may occasionally be exploited. Our basic assumption will be that approximation errors dominate roundoff errors in magnitude in our actual calculations. This can often be achieved, at least in double precision (see Section 1.3 and Example 1.1).

Example 1.1
In this lengthy example we see how discretization errors and roundoff errors both arise in a simple setting.

Consider the problem of approximating the derivative $f'(x_0)$ of a given smooth function $f(x)$ at the point $x = x_0$. For instance, let $f(x) = \sin(x)$

be defined on the real line $-\infty < x < \infty$, and set $x_0 = 1.2$. Thus, $f(x_0) = \sin(1.2) \approx 0.932....$

Further, we consider a situation where $f(x)$ may be evaluated at any point x near x_0, but $f'(x_0)$ may not be directly available, or is computationally prohibitively expensive to evaluate. Thus, we seek ways to approximate $f'(x_0)$ by evaluating f at arguments x near x_0.

A simple minded algorithm may be constructed using *Taylor's series*. For some small, positive value h that we will choose in a moment, write

$$f(x_0 + h) = f(x_0) + hf'(x_0) + \frac{h^2}{2}f''(x_0) + \frac{h^3}{6}f'''(x_0) + \frac{h^4}{24}f''''(x_0) + \cdots .$$

Then,

$$f'(x_0) = \frac{f(x_0 + h) - f(x_0)}{h} - \left(\frac{h}{2}f''(x_0) + \frac{h^2}{6}f'''(x_0) + \frac{h^3}{24}f''''(x_0) + \cdots\right).$$

Our *algorithm* for approximating $f'(x_0)$ is to calculate

$$\frac{f(x_0 + h) - f(x_0)}{h}.$$

The obtained approximation has the *discretization error*

$$\left|f'(x_0) - \frac{f(x_0 + h) - f(x_0)}{h}\right| = \left|\frac{h}{2}f''(x_0) + \frac{h^2}{6}f'''(x_0) + \frac{h^3}{24}f''''(x_0) + \cdots\right|.$$

Geometrically, we approximate the slope of the tangent at the point x_0 by the slope of the chord through neighboring points of f. In Figure 1.2, the tangent is in blue and the chord is in red.

If we know $f''(x_0)$, and it is nonzero, then for h small we can estimate the discretization error by

$$\left|f'(x_0) - \frac{f(x_0 + h) - f(x_0)}{h}\right| \approx \frac{h}{2}\left|f''(x_0)\right|.$$

Even without knowing $f''(x)$ we expect that, provided $f''(x_0) \neq 0$, the discretization error will decrease proportionally to h as h is decreased.

For our particular instance $f(x) = \sin(x)$, we have the exact value $f'(x_0) = \cos(1.2) = 0.362357754476674....$ Carrying out the above algorithm we obtain for $h = 0.1$ the approximation $f'(x_0) \approx (\sin(1.3) - \sin(1.2))/0.1 = 0.31519...$ The *absolute error* (which is the magnitude of the difference between the exact derivative value and the calculated one) thus equals approximately 0.047.

This approximation of $f'(x_0)$ using $h = 0.1$ is not very accurate. We therefore apply the same algorithm using several, smaller and smaller values of h. The resulting errors are as follows:

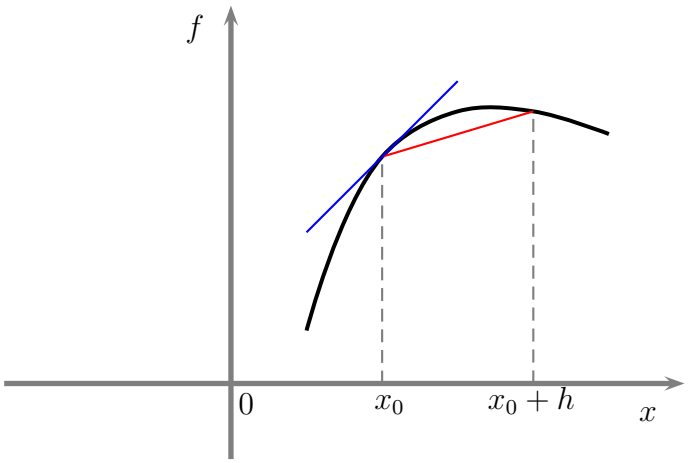

Figure 1.2: A simple instance of numerical differentiation: the tangent $f'(x_0)$ is approximated by the chord $(f(x_0 + h) - f(x_0))/h$.

h	Absolute error
0.1	4.716676e-2
0.01	4.666196e-3
0.001	4.660799e-4
1.e-4	4.660256e-5
1.e-7	4.619326e-8

Indeed, the error appears to decrease like h. More specifically (and less importantly), using our explicit knowledge of $f''(x) = -\sin(x)$, in this case we have that $\frac{1}{2}f''(x_0) \approx -0.466$. The quantity $0.466h$ is seen to provide a rather accurate estimate for the above tabulated error values.

The above calculations, and the ones reported below, were carried out using MATLAB's standard arithmetic. The numbers just recorded might suggest that arbitrary accuracy can be achieved by our algorithm, provided only that we take h small enough. Indeed, suppose we want $\left|\cos(1.2) - \frac{\sin(1.2+h)-\sin(1.2)}{h}\right| < 10^{-10}$. Can't we just set $h \leq 10^{-10}/0.466$ in our algorithm?

Not quite! Let us record results for very small, positive values of h:

h	Absolute error
1.e-8	4.361050e-10
1.e-9	5.594726e-8
1.e-10	1.669696e-7
1.e-11	7.938531e-6
1.e-13	6.851746e-4
1.e-15	8.173146e-2
1.e-16	3.623578e-1

A log-log plot of the error vs h is provided in Figure 1.3. We can clearly

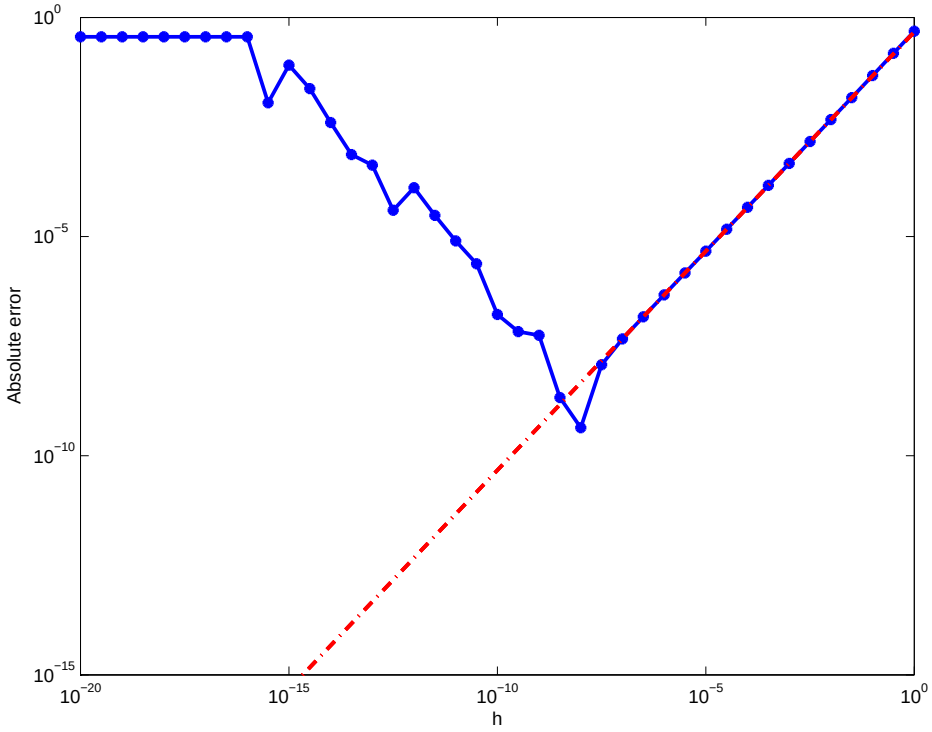

Figure 1.3: The combined effect of discretization and roundoff errors. The solid curve interpolates the computed values of $|f'(x_0) - \frac{f(x_0+h)-f(x_0)}{h}|$ for $f(x) = \sin(x),\ x_0 = 1.2$. Shown in dash-dot style is a straight line depicting the discretization error without roundoff error.

see that, as h is decreased, at first (from right to left in the figure) the error decreases along a straight line, but this trend is altered and eventually reversed. The MATLAB script which generates the plot in Figure 1.3 is

```
x0 = 1.2;
f0 = sin(x0);
fp = cos(x0);
i = -20:0.5:0;
h = 10.^i;
err = abs (fp - (sin(x0+h) - f0)./h );
d_err = f0/2*h;
loglog (h,err,'-*');
hold on
loglog (h,d_err,'r-.');
xlabel('h')
ylabel('Absolute error')
```

The reason for the error "bottoming out" at about $h = 10^{-8}$ is that the total error consists of contributions of both discretization and roundoff errors. The discretization error decreases in an orderly fashion as h decreases, and it dominates the roundoff error when h is relatively large. But when h gets below the approximate value 10^{-8} the discretization error becomes very small and roundoff error starts to dominate (i.e., it becomes larger in magnitude). The roundoff error has a somewhat erratic behaviour, as is evident from the small oscillations that are present in the graph in a couple of places. Overall, the roundoff error *increases* as h decreases. This is one reason why we want it always dominated by the discretization error when approximately solving problems involving numerical differentiation, such as differential equations for instance.

We will later understand the precise source of the roundoff error exhibited in this case: See Section 1.3, under the header 'Roundoff error accumulation and cancellation errror', and Exercise 2.

♦

Relative and absolute errors

In Example 1.1 we have recorded measured values of absolute error. In fact, there are in general two basic types of measured error: Given a quantity u and its approximation v,

- the *absolute error* in v is $|u - v|$, and
- the *relative error* (assuming $u \neq 0$) is $\frac{|u-v|}{|u|}$.

The relative error is usually a more meaningful measure [1]. This is especially true for errors in floating point representation, a point to which we return in

[1] However, there are exceptions, especially when the approximated value is small in magnitude – let us not worry about this yet.

Section 1.3. For example, we record absolute and relative errors for various hypothetical calculations in the following table:

u	v	Absolute Error	Relative Error
1	0.99	0.01	0.01
1	1.01	0.01	0.01
-1.5	-1.2	0.3	0.2
100	99.99	0.01	0.0001
100	99	1	0.01

Evidently, when $|u| \approx 1$ there is not much difference between absolute and relative error measures. (This is the case in Example 1.1 above.) But when $|u| \gg 1$, the relative error is more meaningful. In particular, writing for the last row of this table $u = 1 \times 10^2$, $v = 0.99 \times 10^2$, we expect to be able to work with such an approximation as accurately as with the one in the first row, and this is borne out by the value of the relative error.

Algorithm properties

Since we must live with errors in our numerical computations, the next natural question is regarding appraisal of a given computed solution: In view of the fact that the problem and the numerical algorithm both yield errors, can we trust the numerical solution of a nearby problem (or the same problem with slightly different data) to differ by only a little from our computed solution? A negative answer could make our computed solution meaningless![2]

This question can be complicated to answer in general, and it leads to notions such as **problem sensitivity** and **algorithm stability**. If the problem is too sensitive, or **ill-conditioned**, meaning that even a small perturbation in the data produces a large difference in the result, then no algorithm may be found for that problem which would meet our requirement of solution robustness. See Figure 1.4. Some modification in the problem definition may be called for in such cases.

For instance, the problem of numerical differentiation depicted in Example 1.1 turns out to be ill-conditioned when extreme accuracy (translating to very small values of h) is required.

The job of a **stable** algorithm for a given problem is to yield a numerical solution which is the exact solution of an only slightly perturbed problem.

[2]Here we refer to intuitive notions of "large" vs "small" quantities and of values being "close to" vs "far from" one another. While these notions can be quantified and thus made more precise, this would typically make definitions cumbersome and harder to understand.

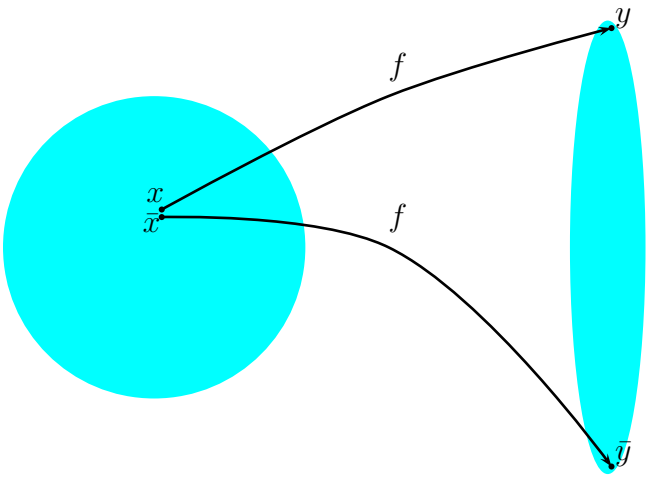

Figure 1.4: An ill-conditioned problem of computing $y = f(x)$: When the input x is slightly perturbed to $\bar{x}$, the result $\bar{y} = f(\bar{x})$ is far from y. If the problem were well-conditioned, we would be expecting the distance between y and $\bar{y}$ to be more comparable in magnitude to the distance between x and $\bar{x}$.

See Figure 1.5. Thus, if the algorithm is stable and the problem is **well-conditioned** (i.e., not ill-conditioned) then the computed result $\bar{y}$ is close to the exact y.

In general, it is impossible to prevent *linear* accumulation of roundoff errors during a calculation, and this is acceptable if the linear rate is moderate (i.e., the constant c_0 below is not very large). But we must prevent *exponential* growth! Explicitly, if E_n measures the relative error at the nth operation of an algorithm, then

$$\begin{aligned} E_n &\simeq c_0 n E_0 \text{ represents linear growth;} \\ E_n &\simeq c_1^n E_0 \text{ represents exponential growth,} \end{aligned}$$

for some constants c_0 and $c_1 > 1$.

An algorithm exhibiting relative exponential error growth is *unstable*. Such algorithms must be avoided!

Example 1.2
Consider evaluating the integrals

$$y_n = \int_0^1 \frac{x^n}{x+10} dx$$

for $n = 1, 2, \ldots, 30$.

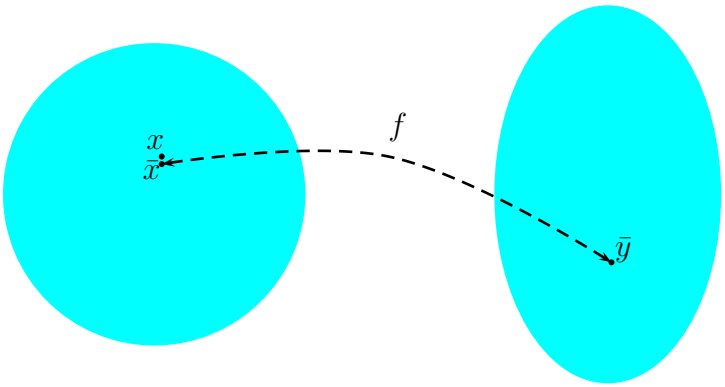

Figure 1.5: A stable algorithm for computing $y = f(x)$: the output $\bar{y}$ is the exact result, $\bar{y} = f(\bar{x})$, for a slightly perturbed input; i.e., $\bar{x}$ which is close to the input x. Thus, if the algorithm is stable and the problem is well-conditioned then the computed result $\bar{y}$ is close to the exact y.

Observe at first that analytically,

$$y_n + 10y_{n-1} = \int_0^1 \frac{x^n + 10x^{n-1}}{x+10} dx = \int_0^1 x^{n-1} dx = \frac{1}{n}.$$

Also,

$$y_0 = \int_0^1 \frac{1}{x+10} dx = \ln(11) - \ln(10).$$

An algorithm which may come to mind is therefore as follows:

- Evaluate $y_0 = \ln(11) - \ln(10)$.
- For $n = 1, \ldots, 30$ evaluate

$$y_n = \frac{1}{n} - 10\ y_{n-1}.$$

Note that applying this recursion formula would give *exact* values if floating point errors were not present.

However, this algorithm is obviously unstable, as the magnitude of roundoff errors gets multiplied by 10 (like compound interest on a shark loan, or so it feels) each time the recursion is applied. Thus, there is exponential error growth with $c_1 = 10$. In MATLAB (which automatically employs IEEE double precision floating point arithmetic, see Section 1.3) we obtain $y_0 = 9.5310e{-}02$, $y_{18} = -9.1694e{+}01$, $y_{19} = 9.1694e{+}02, \ldots, y_{30} = -9.1694e{+}13$.

Note that the exact values all satisfy $0 < y_n < 1$ (Exercise: why?). Thus, the computed solution, at least for $n \geq 18$, is meaningless!

♦

Thankfully, such extreme instances of instability as depicted in Example 1.2 will not occur in any of the algorithms developed in these notes from here on.

An assessment of the usefulness of an algorithm may be based on a number of criteria:

- **Accuracy**

  This issue is intertwined with the issue of error types and was discussed at the start of this section and in Example 1.1. The important point is that the accuracy of a numerical algorithm is an important parameter in its assessment, and when designing numerical algorithms it is necessary to be able to point out what magnitude of error is to be expected when the algorithm is carried out.

- **Efficiency**

  This depends on speed of execution in terms of CPU time and storage space requirements. Details of an algorithm implementation within a given computer language and an underlying hardware configuration may play an important role in yielding an observed code efficiency.

  Often, though, a machine-independent estimate of the number of floating-point operations (**flops**) required gives an idea of the algorithm's efficiency.

  Example 1.3
  A polynomial of degree n,

  $$p_n(x) = c_0 + c_1 x + \ldots + c_n x^n$$

  requires $\mathcal{O}(n^2)$ operations to evaluate at a fixed point x, if done in a brute force way without intermediate storing of powers of x. But using the *nested form*, also known as Horner's rule,

  $$p_n(x) = (\cdots((c_n x + c_{n-1})x + c_{n-2})x \cdots)x + c_0$$

  suggests an evaluation algorithm which requires only $\mathcal{O}(n)$ operations, i.e. requiring linear instead of quadratic time. A MATLAB script for nested evaluation is as follows:

```
% Assume the polynomial coefficients are already stored
% in array c such that for any real x,
```

```
% p(x) = c(1) + c(2)x + c(3)x^2 + ... + c(n+1)x^n
p = c(n+1);
For j = n:-1:1
  p = p*x + c(j);
end
```

♦

It is important to realize that while operation counts like this one often give a rough idea of algorithm efficiency, they do not give the complete picture regarding execution speed, as they do not take into account the price (speed) of memory access which may vary considerably. Moreover, any setting of parallel computing is ignored in a simple operation count as well. Curiously, this is part of the reason why the MATLAB command `flops`, which was an integral part of this language for many years, was removed from MATLAB 6. Indeed, in modern computers, cache access, blocking and vectorization features, and other parameters are crucial in the determination of execution time. Those, unfortunately, are much more difficult to assess compared to operation count, and we will not get in this text into the gory details of the relevant, important issues.

Other theoretical properties yield indicators of efficiency, for instance the **rate of convergence**. We return to this in later chapters.

- **Robustness**

  Often, the major effort in writing **numerical software**, such as the routines available in MATLAB for function approximation and integration, is spent not on implementing the essence of an algorithm to carry out the given task but on ensuring that it would work under all weather conditions. Thus, the routine should either yield the correct result to within an acceptable error tolerance level, or it should fail gracefully (i.e., terminate with a warning) if it does not succeed to guarantee a "correct result".

  There are intrinsic numerical properties that account for the robustness and reliability of an algorithm. Chief among these is the rate of accumulation of errors. In particular, *the algorithm must be stable*, as we have already seen in Example 1.2.

1.3 Roundoff errors and computer arithmetic

As we have seen, various errors may arise in the process of calculating an approximate solution for a mathematical model. Here we concentrate on one

Note: Readers who do not require detailed knowledge of roundoff error and its propagation during a computation may skip the rest of this chapter, at least upon first reading, provided that they accept the notion that each number representation and each elementary operation (such as + or $*$) in standard floating point arithmetic introduces a (small) relative error: up to about $\eta = 2.2e-16$ in today's standard floating point systems.

error type, *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.

Figure 1.6: The Patriot Missile catastrophically missed its target due to rounding errors.

Example 1.4
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 disasters. One of the more spectacular disasters was the Patriot Missile (Figure 1.6) failure in Dharan, Saudi Arabia, on February 25, 1991, which resulted in 28 deaths. This failure was ultimately traced back to poor handling of rounding errors in the missile's software. A web site maintained by D. Arnold,
`http://www.ima.umn.edu/~arnold/disasters/disasters.html`
contains the details of this story and others. For a larger collection of software bugs, see
`http://wwwzenger.informatik.tu-muenchen.de/persons/huckle/bugse.html`
♦

We discuss the following topics in this section:

- Floating point numbers
- Errors in floating point representation
- Roundoff error accumulation and cancellation error
- The rough appearance of roundoff error
- Machine representation and the IEEE standard
- Floating point arithmetic
- Errors in a general floating point representation

Floating point numbers

Any real number is accurately representable by an infinite decimal sequence of digits.[3] For instance,

$$\frac{8}{3} = 2.6666\ldots = \left(\frac{2}{10^1} + \frac{6}{10^2} + \frac{6}{10^3} + \frac{6}{10^4} + \frac{6}{10^5} + \cdots\right) \times 10^1.$$

This is an infinite series, but computers use a finite amount of memory to represent real numbers. Thus, only a finite number of digits may be used to represent any number, no matter by what representation method.

For instance, we can chop the infinite decimal representation of 8/3 after 4 digits,

$$\frac{8}{3} \simeq \left(\frac{2}{10^1} + \frac{6}{10^2} + \frac{6}{10^3} + \frac{6}{10^4}\right) \times 10^1 = 0.2666 \times 10^1.$$

Generalizing this, we can speak of t decimal digits (and call t the *precision*). For each real number x we associate a *floating point representation*, denoted $\mathrm{fl}(x)$, given by

$$\begin{aligned} \mathrm{fl}(x) &= \pm 0.d_1 d_2 \ldots d_{t-1} d_t \times 10^e \\ &= \pm\left(\frac{d_1}{10^1} + \frac{d_2}{10^2} + \cdots + \frac{d_{t-1}}{10^{t-1}} + \frac{d_t}{10^t}\right) \times 10^e. \end{aligned}$$

The sign, digits d_i and exponent e are chosen so that $\mathrm{fl}(x)$ closely approximates the value of x: shortly we will discuss how close that is. In the above example, $t = 4$ and $e = 1$.

[3] 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.

The above representation is not unique; for instance,

$$0.2666 \times 10^1 = 0.02666 \times 10^2.$$

Therefore, we **normalize** the representation by insisting that $d_1 \neq 0$. Thus,

$$1 \leq d_1 \leq 9, \;\; 0 \leq d_i \leq 9, \;\; i = 2, \ldots, t.$$

Not only the precision is limited to a finite number of digits, also the range of the exponent must be restricted. Thus, there are integers $U > 0$ and $L < 0$ such that all eligible exponents in a given floating point system satisfy

$$L \leq e \leq U.$$

The largest number precisely representable in such a system is

$$0.99 \ldots 99 \times 10^U \lessapprox 10^U,$$

and the smallest positive number is

$$0.10 \ldots 00 \times 10^L = 10^{L-1}.$$

In addition, there are certain numbers that cannot be represented in a normalized fashion but are necessary in any floating point system. One such number is the celebrated 0... Numbers like 0 and ∞ are represented as special combinations of bits, according to an agreed upon convention for the given floating point system. A specific example is provided a little later in this section, for the IEEE standard.

Example 1.5
Consider a (toy) decimal floating point system as above with $t = 4$, $U = 2$ and $L = -1$. The decimal number 2.666 is precisely representable in this system because $L < e < U$.

The largest number in this system is 99.99, the smallest is -99.99, and the smallest positive number is $10^{-2} = 0.01$.

How many different numbers are in this floating point system? 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 $9 \times 10 \times 10 \times 10 = 9000$ different normalized fractions possible. The exponent can be one of $U - L + 1 = 4$ values, so in total there are $4 \times 9000 = 36000$ different positive numbers possible. There is the same total of negative numbers, and then there is the number 0. So, there are 72001 different numbers in this floating point system.

♦

Errors in floating point representation

How accurate is a floating point representation of the real numbers? First, recall how we measure errors. For a given floating point system, denoting by $\mathrm{fl}(x)$ the floating point number that approximates x, the absolute and relative errors are given by

$$\begin{aligned} \text{Absolute error } &= |\mathrm{fl}(x) - x|; \\ \text{Relative error } &= \frac{|\mathrm{fl}(x) - x|}{|x|}. \end{aligned}$$

As was pointed out in Section 1.2, the relative error is generally a more meaningful measure in floating point representation, because it is independent of a change of exponent. Thus, if $u = 1,000,000 = 1 \times 10^6$, $v = \mathrm{fl}(u) = 990,000 = 0.99 \times 10^6$, we expect to be able to work with such an approximation as accurately as with $u = 1$, $v = \mathrm{fl}(u) = 0.99$, and this is borne out by the value of the relative error.

There are two distinct ways to do the "cutting" to store a real number $x = \pm(0.d_1d_2d_3 \ldots d_td_{t+1}d_{t+2} \ldots) \times 10^e$ using only t digits.

- **Chopping**: ignore digits $d_{t+1}, d_{t+2}, d_{t+3} \ldots$.
- **Rounding**: add 1 to d_t if $d_{t+1} \geq \frac{10}{2} = 5$, then ignore digits $d_{t+1}, d_{t+2}, d_{t+3} \ldots$.

Here is a simple example, where we chop and round with $t = 3$:

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

Let $x \mapsto \mathrm{fl}(x) = 0.f \times 10^e$, where f is obtained as above by either chopping or rounding. Then, the absolute error committed in using a machine representation of x is bounded by

$$\text{absolute error} \leq \begin{cases} 10^{-t} \cdot 10^e, & \text{for chopping} \\ \frac{1}{2}10^{-t} \cdot 10^e, & \text{for rounding} \end{cases}.$$

(This is not difficult to see. In any case, we show it more generally later on, in the last subsection.)

Given these bounds on the absolute error, we now bound the relative error. Note that due to normalization,

$$|x| \geq \left(\frac{1}{10} + \frac{0}{10^2} + \cdots + \frac{0}{10^t} + \cdots \right) \times 10^e = 0.1 \times 10^e.$$

For chopping, then,

$$\frac{|\mathrm{fl}(x) - x|}{|x|} \leq \frac{10^{-t} \cdot 10^e}{0.1 \cdot 10^e} = 10^{1-t} =: \eta.$$

For rounding, like the case for absolute error, the relative error is half of what it is for chopping, hence

$$\eta = \frac{1}{2} 10^{1-t}.$$

For the toy floating point system of Example 1.5, $\eta = \frac{1}{2}10^{1-4} = 0.0005$.

The quantity η is fundamental, as it expresses a tight bound on the 'atomic' relative error we may expect to result from each elementary operation with floating point numbers. It has been called **machine precision**, **rounding unit**, **machine epsilon** and more. Furthermore, the negative of its exponent, $t-1$ (for the rounding case), is often referred to as the **number of significant digits**.

Roundoff error accumulation and cancellation error

As 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 add a small relative error; but how do these errors accumulate? In general, as we have already mentioned, error growth which is linear in the number of operations is unavoidable. Yet, there are a few things to watch out for.

- Cautionary notes:

  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 has 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**). We return to this type of error below.

- 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 non-fatal: the system usually sets the number to 0 and continues. (MATLAB does this, quietly.)

`NaN` is a combination of letters that stands for 'not a number', which naturally one dreads to see in one's calculations. 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 inadvertantly encounter a few `NaN`'s before this course is over.

Let us give an example to show how overflow may sometimes be avoided.

Example 1.6
Consider computing $c = \sqrt{a^2 + b^2}$ in a floating point system with 4 decimal digits and 2 exponent digits, for $a = 10^{60}$ and $b = 1$. The correct result in this precision is $c = 10^{60}$. But overflow results during the course of calculation. (Why?) Yet, this overflow can be avoided if we rescale ahead of time: Note $c = s\sqrt{(a/s)^2 + (b/s)^2}$ 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 here.

♦

Cancellation error deserves a special attention, because it often appears in practice in an identifiable way. For instance, recall Example 1.1. 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,

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

with a small step h, then there is a cancellation error in the numerator, which is then magnified by the denominator.

Sometimes such errors can be avoided by a simple modification in the algorithm. An instance is illustrated in Exercise 1. Here is another one.

Example 1.7
Suppose we wish to compute $y = \sqrt{x+1} - \sqrt{x}$ for $x = 100000$ in a 5-digit decimal arithmetic. Clearly, the number 100001 cannot be represented in this floating point system exactly, and its representation in the system (regardless of whether chopping or rounding is used) is 100000. 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 (in 5-digit decimal arithmetic) yields $.15811 \times 10^{-2}$, which happens to be the correct value to 5 decimal digits. The computation is carried out by a straightforward utilization of the following routine, that returns a number x, rounded to n decimal digits.

```
function y = roundc(x,n)
%
% function y = roundc(x,n)
%
% Returns x rounded to n decimal digits
%         (take n < number of significant digits in x )
% Note x can be an array; y would have the same size

xx = abs(x)+1.e-20;
e = ceil(log10(xx));    % number's decimal exponent
f = xx ./ 10.^e;        % number's fraction

s = 1;
frac = zeros(size(x));
for j = 1:n
  s = s*10;
  d = floor(f*s + 0.5); % extract digit
  f = f - d/s;
  frac = frac + d/s;    % add digit to rounded fraction
end

y = sign(x) .* frac .* 10.^e;
```

♦

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

Example 1.8
Suppose we wish to compute

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

for a small value of $x > 0$, $|x| \ll 1$. Directly using the above 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

$$\sinh x = x + \frac{x^3}{6} + \frac{x^5}{120} + \dots$$

may prove useful. For example, the cubic approximation $x + \frac{x^3}{6}$ should be an effective (and computationally inexpensive) approximation, if x is sufficiently small. Clearly, this formula is not prone to cancellation error. Moreover, it is easy to see that the discretization error in this approximation is $\approx \frac{x^5}{120}$ for $|x| \leq 1$.

Employing the same 5-digit decimal arithmetic as used in Example 1.7 for our cubic approximation, we compute $\sinh(0.1) = 0.10017$ and $\sinh(0.01) = 0.01$. These are the 'exact' values in this floating point system. On the other hand, employing 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. ♦

Some insight into the damaging effect of subtracting two nearly equal numbers in a floating point system can be observed by examining a bound for 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

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

Although the right hand side is just a bound, it could be a tight one, since it merely depends on the floating point system's ability to represent the numbers x and y. However, the denominator is very close to zero if $x \approx y$ regardless of how well the floating point system can do, and so the relative error could become very large.

A note of caution is in place with regard to any suggested remedies for avoiding cancellation errors. In large scale computations it is not practical to repeatedly rewrite formulas or use Taylor expansions and expect the trouble to disappear. It is nevertheless worthwhile keeping those seemingly naive techniques in mind. In some cases, even if the computation is long and intensive, it may be based on a short algorithm, and a simple change of a formula may greatly improve the algorithm's stability.

The rough appearance of roundoff error

Let us study the seemingly unstructured behaviour of roundoff error as becomes apparent, for instance, already in Figure 1.3.

Example 1.9
We examine the behaviour of roundoff errors by evaluating $\sin(2\pi t)$ at 101 equidistant points between 0 and 1, rounding these numbers to 5 decimal digits, and plotting the differences. This is conveniently achieved by the following MATLAB instructions:

```
t = 0 : .01 : 1;
tt = sin(2 * pi * t);
rt = roundc(tt, 5);
round_err = tt − rt;
plot(t, round_err);
title ('roundoff error in sin(2πt) rounded to 5 decimal digits')
xlabel('t')
ylabel('roundoff error')
```

The function `roundc` has been introduced in Example 1.7. The result is depicted in Figure 1.7. Note the disorderly, "high frequency" oscillation of the roundoff error. This is in marked contrast with discretization errors, which are usually "smooth", as we have seen in Example 1.1 (note the straight line drop of the error in Figure 1.3 for relatively large h where the discretization error dominates).

♦

Machine representation and the IEEE standard

Note: Here is where we get more technical in the present chapter.

Of course, computing machines do not necessarily use base 10 (especially those machines that do not have 10 fingers on their hands). Because many humans are used to base 10 we have used it until now in this section in order to introduce and demonstrate various concepts. But computer storage is in an integer multiple of bits, hence typical computer representations use bases that are powers of 2. In general, if the base is β (i.e., a digit can have values

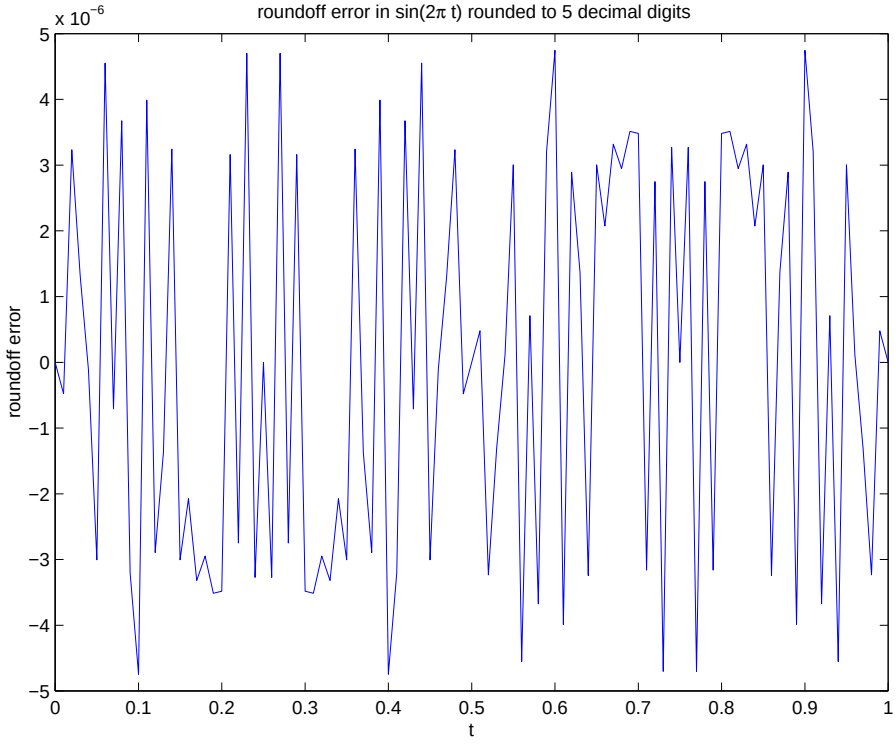

Figure 1.7: The "almost random" nature of roundoff errors.

from 0 to $\beta-1$) then the above derivations can be repeated with β replacing the decimal base 10. In particular, let us frame the general expression for the all important machine presicion constant η in rounding arithmetic. It is derived in the last subsection below.

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

The common base for most computers today, following the *IEEE standard* set in 1985, is base 2. In this base each digit d_i may only have two values, 0 or 1. Thus, the first (normalized) digit must be $d_1 = 1$ and need not be stored. The IEEE standard therefore also shifts $e \leftarrow e - 1$, so that actually,

$$\mathrm{fl}(x) = \pm\left(1 + \frac{d_2}{2} + \frac{d_3}{4} + \cdots + \frac{d_t}{2^{t-1}}\right) \times 2^e.$$

The standard floating point word in the IEEE standard (in which for instance the computations of Examples 1.1 and 1.2 were carried out) requires

64 bits of storage and is called **double precision** or *long word*. Of these 64 bits, one is allocated for sign s (the number is negative iff $s = 1$), 11 for the exponent and 52 for the fraction:

Double precision (64 bit word)

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

$\beta = 2$, $t = 52 + 1$, $L = -1023$, $U = 1024$

Since the fraction f contains $t - 1$ digits the precision is $t = 52 + 1 = 53$. Thus, a given bit pattern in the last 52 bits of a long word is interpreted as a sequence of binary digits $d_2 d_3 \cdots d_{53}$. Since $2^{11} = 2048$ different exponents can be distinguished the upper and lower limits are set at $U = 1024$, $L = -1023$. (Note $U - L + 1 = 2048$.) Therefore, 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 L, i.e., $e = b - 1023$.

Example 1.10
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, $b = 1\times2^{10}+1\times2^2+1\times2^1+1\times2^0 = 1031$. 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

$$s \cdot (1 + f) \times 2^e = 1.90625 \times 2^8 = 488.$$

♦

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

$\beta = 2,\ t = 23 + 1,\ L = -127,\ U = 128$

The IEEE standard reserves the endpoints of the exponent range for special purposes, and allows only exponents e satisfying $L < e < U$. The largest number precisely representable in a long word is therefore

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

and the smallest positive number is

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

How does the IEEE standard store 0 and ∞? 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 $b = 1 \cdots 1, f = 1 \cdots 1$;
- The pattern $b = 1 \cdots 1,\ f \neq 1 \cdots 1$ is by convention `NaN`.

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

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

and the smallest positive number is

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

The formulae for the machine precision or rounding unit η introduced above in base 10 remain the same, except of course that here the base changes to $\beta = 2$. Using the word arrangement for single and double precision, η is calculated as follows:

- For single precision with chopping, $\eta = \beta^{1-t} = 2^{1-24} = 1.19 \times 10^{-7}$ (so, there are 23 significant binary digits, or about 7 decimal digits).
- For double precision with chopping, $\eta = \beta^{1-t} = 2^{1-53} = 2.2 \times 10^{-16}$ (so, there are 52 significant binary digits, or about 16 decimal digits).

In MATLAB typing `eps` (without explicitly assigning it a value beforehand) displays approximately the value $2.22e-16$.

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 a very high precision (e.g. in semiconductor simulation, numerical relativity, astronomical calculations).

Floating point arithmetic

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. We will not dwell on this much, except to say that 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 was rounded to the nearest floating point 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 1.11
Consider a floating point system with decimal base $\beta = 10$ and four digits $t = 4$. Thus, the rounding unit is $\eta = \frac{1}{2} \times 10^{-3}$. Let

$$x = .1103, \quad y = .9963 \times 10^{-2}.$$

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

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

However, if we were to subtract these two numbers without guard digits we would obtain $.1103 - .0099 = .1004$. Now,

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

♦

Example 1.12
Generally, $\mathrm{fl}(1+\alpha) = 1$ for any number α which satisfies $|\alpha| \le \eta$. In particular, the following MATLAB commands

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

produce the output $eta = 1.1102e-16$, $beta = 0$. Returning to Example 1.1 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. ♦

Errors in a general floating point representation

A general **floating point system** may be defined by four values (β, t, L, U) where

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

For each $x \in \mathbb{R}$ (i.e., for each real number x), there is an associated floating point representation

$$\mathrm{fl}(x) = \pm \left(\frac{d_1}{\beta^1} + \frac{d_2}{\beta^2} + \cdots + \frac{d_t}{\beta^t} \right) \times \beta^e,$$

where d_i are integer digits in the range $0 \le d_i \le \beta - 1$. The number $\mathrm{fl}(x)$ is an approximation of x, in general. To ensure uniqueness of this representation it is normalized to satisfy $d_1 \neq 0$ by adjusting the exponent e so that leading zeros are dropped. (However, unless $\beta = 2$ this does not fix the value of d_1.) Moreover, e must be in the range $L \le e \le U$.

Obviously, the largest number that can be precisely expressed in this system is obtained by setting $d_i = \beta - 1, i = 1, \ldots, t$, and $e = U$. The smallest positive number is obtained by setting $d_1 = 1$, then $d_i = 0, i = 2, \ldots, t$, and $e = L$.

The definitions of chopping and rounding extend directly for a more general floating point system. To store a real number $x = \pm(.d_1 d_2 d_3 \ldots d_t d_{t+1} d_{t+2} \ldots) \cdot \beta^e$ using only t digits,

- chopping ignores digits $d_{t+1}, d_{t+2}, d_{t+3} \ldots$; whereas
- rounding adds 1 to d_t if $d_{t+1} \ge \frac{\beta}{2}$.

The IEEE standard uses a mixture of both ("unbiased rounding").

Let $x \mapsto \mathrm{fl}(x)=0.f \times \beta^e$. (For the IEEE standard, there is a slight variation, which is immaterial here.) Then, the absolute error committed in using a machine representation of x is

$$\text{absolute error} \le \begin{cases} \beta^{-t} \cdot \beta^e, & (1 \text{ in the last digit for chopping}) \\ \frac{1}{2}\beta^{-t} \cdot \beta^e, & (\frac{1}{2} \text{ in the last digit for rounding}) \end{cases}.$$

Proof:

Let us show these bounds, for those who just would not believe us otherwise. We also use the opportunity to take a closer look at the structure of the generated representation error. Assume for simplicity of notation that $x > 0$, and write (without being fussy about limits and irrational numbers)

$$x = (.d_1 d_2 d_3 \ldots d_t d_{t+1} d_{t+2} \ldots) \cdot \beta^e = \left(\sum_{i=1}^{\infty} \frac{d_i}{\beta^i} \right) \beta^e.$$

- Consider chopping. Then,

  $$\mathrm{fl}(x) = (.d_1 d_2 d_3 \ldots d_t) \cdot \beta^e = \left(\sum_{i=1}^{t} \frac{d_i}{\beta^i} \right) \beta^e.$$

  So, $\beta^{-e}[x - \mathrm{fl}(x)] = \sum_{i=t+1}^{\infty} \frac{d_i}{\beta^i}$, and we ask how large this can be. Each digit satisfies $d_i \le \beta - 1$, so at worst

  $$\beta^{-e}[x - \mathrm{fl}(x)] \le (\beta - 1) \sum_{i=t+1}^{\infty} \frac{1}{\beta^i} = \beta^{-t}.$$

  Note, incidentally, that if $x > 0$ then always $\mathrm{fl}(x) < x$, and if $x < 0$ then always $\mathrm{fl}(x) > x$. This is not very good statistically, and does not occur with rounding.

- Consider rounding. Now

  $$\mathrm{fl}(x) = \begin{cases} (.d_1 d_2 d_3 \ldots d_{t-1} d_t) \cdot \beta^e = \left(\sum_{i=1}^{t} \frac{d_i}{\beta^i} \right) \beta^e & \text{if } d_{t+1} < \beta/2 \\ (.d_1 d_2 d_3 \ldots d_{t-1} [d_t + 1]) \cdot \beta^e = \left(\frac{1}{\beta^t} + \sum_{i=1}^{t} \frac{d_i}{\beta^i} \right) \beta^e & \text{if } d_{t+1} \ge \beta/2. \end{cases}$$

  So, if $d_{t+1} < \beta/2$ then

  $$\beta^{-e}[x - \mathrm{fl}(x)] \le \frac{\beta/2 - 1}{\beta^{t+1}} + (\beta - 1) \sum_{i=t+2}^{\infty} \frac{1}{\beta^i} = \frac{\beta/2 - 1}{\beta^{t+1}} + \frac{1}{\beta^{t+1}} = \frac{1}{2}\beta^{-t},$$

and if $d_{t+1} \geq \beta/2$ then

$$\begin{aligned} |\beta^{-e}[x - \mathrm{fl}(x)]| &= \left| \sum_{i=t+1}^{\infty} \frac{d_i}{\beta^i} - \frac{1}{\beta^t} \right| = \frac{1}{\beta^t} - \sum_{i=t+1}^{\infty} \frac{d_i}{\beta^i} \\ &\leq \frac{1}{\beta^t} - \frac{\beta/2}{\beta^{t+1}} = \frac{1}{2}\beta^{-t}. \end{aligned}$$

♦

Having established the bounds for absolute errors, let us consider next the relative errors in floating point number representations. Since the fraction is normalized we have

$$|x| \geq 1/\beta \times \beta^e.$$

For chopping, the relative error is therefore bounded by

$$\frac{|\mathrm{fl}(x) - x|}{|x|} \leq \frac{\beta^{-t} \cdot \beta^e}{\beta^{-1} \cdot \beta^e} = \beta^{1-t} =: \eta.$$

For rounding, the relative error is half of what it is for chopping (as for the absolute error), so

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

This is the *machine precision* or *rounding unit*.

If you think of how a given floating point system represents the real line you'll find that it has a somwhat 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. See Exercise 7.

Most annoying 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! To address this the IEEE standard introduces into the floating point system next to 0 additional, *subnormal numbers* which are not normalized. We will leave it at that.

Finally, we note that when using rounding arithmetic, errors tend to be more random in sign than when using chopping. Statistically, this gives a higher chance for occasional error cancellations along the way. We will not get into this further here.

1.4 Exercises

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

   b) 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 $\delta = 1.e-11$.

   c) Explain the difference in the results of the two calculations.

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 $h = 1e-20, 1e-19, \ldots, 1$. Compare your results to the results presented in Example 1.1.

   c) Explain the difference in accuracy between your results and the results reported in Example 1.1.

3. The Stirling approximation

$$S_n = \sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^n$$

   is used to approximate $n! = 1 \cdot 2 \cdot \cdots \cdot n$, for large n. Here $e = \exp(1) = 2.7182818...$
   Write a program that computes and displays $n!$ and S_n, as well as the relative and absolute errors, for $1 \leq n \leq 20$.

   Explain the meaning of your results.

4. Consider again the problem presented in Example 1.2, namely, computing the integrals

$$y_n = \int_0^1 \frac{x^n}{x+10} dx$$

for $n = 1, 2, \ldots, 30$. There we saw a numerically unstable procedure for carrying out the task.

Derive a formula for approximately computing these integrals based on evaluating y_{n-1} given y_n. Show that for any given value $\varepsilon > 0$ and n_0, there exists n_1 such that taking $y_{n_1} = 0$ as a starting value will produce integral evaluations y_n with an absolute error smaller than ε for all $0 < n \leq n_0$. Explain why your algorithm is stable and write a MATLAB function that computes the value of y_{20} within an absolute error of at most 10^{-5}.

5. 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.)

6. 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 1.3, the company's floating point system is specified by (β, t, L, U). You may assume that:

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

Questions:

a) How many different nonnegative floating point values can be represented by this floating point system? (Any answer within 5 of the correct one is fine.)

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?

7. If you generate in MATLAB a row vector x containing all the floating point numbers in a given system, another row vector y of the same dimension as x containing $0's$, and then plot discrete values of y vs x using the symbol $'+'$, you'll get a picture of sorts of the floating point number system. The relevant commands for such a display are

```
y = zeros(1,length(x));
plot (x,y,'+')
```

Produce such a plot for the system $(\beta, t, L, U) = (2, 3, -2, 3)$. (Do not assume the IEEE special conventions.) What do you observe? Also, calculate the rounding unit for this modest floating point system.

8. (a) The number $\frac{8}{3} = 2.6666\ldots$ 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 precition t) in which this number does have an exact representation? If yes then describe one such.

   (b) Same question for the irrational number π.

9. The roots of the quadratic equation

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

with $b^2 > c$ are given by

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

Note $x_1 x_2 = c$.

The following MATLAB scripts calculate these roots using two different algorithms:

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

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

Which algorithm gives a more accurate result in general? Choose one of the following options.

(a) Algorithm (i)

(b) Algorithm (ii)

(c) Both algorithms produce the same result

Justify your choice in one short sentence. (NB The point is to answer this question without any computing.)

10. Write a MATLAB program which will:

    (a) Sum up $1/n$ for $n = 1, 2, \ldots, 10000$.

    (b) Round each number $1/n$ to 5 decimal digits, and then sum them up *in* 5-*digit decimal arithmetic* for $n = 1, 2, \ldots, 10000$.

    (c) Sum up the same rounded numbers (in 5-digit decimal arithmetic) in reverse order, i.e. for $n = 10000, \ldots, 2, 1$.

    Compare the three results and explain your observations.

11. 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 flop counts? (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? (Justify briefly.)

    (c) Write a little MATLAB script to check whether your answer to the previous question was correct.

12. 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, $\mathrm{fl}(x+y) = (x+y)(1+\epsilon)$, $|\epsilon| \leq \eta$. 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.

1.5 Additional notes

- Here are some internet introdutions to MATLAB. There are many more:
  - Kermit Sigmon, MATLAB tutorial,

    `http://www.mines.utah.edu/gg_computer_seminar/matlab/matlab.html`
  - Ian Cavers, Introductory guide to MATLAB,

    `http://www.cs.ubc.ca/spider/cavers/MatlabGuide/guide.html`
  - Mark Gockenbach, A practical introduction to MATLAB,

    `http://www.math.mtu.edu/~msgocken/intro/intro.html`
- A lot of thinking went in the early days 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 involved 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 web page for various interesting related documents:

  `http://www.cs.berkeley.edu/~wkahan/`

  A short, accessible textbook which discusses IEEE floating point in great detail is Overton [30]. A comprehensive and thorough treatment of roundoff errors and many aspects of numerical stability can be found in Higham [23].
- It may be surprising for the reader to know how much attention has been given throughout the years to the definition of scientific computing and/or numerical analysis. Early definitions made use of terms like roundoff errors and other terms, which were later perceived as not

very appealing. A very nice account of the evolution of this seemingly trivial but nevertheless important issue, can be found in Trefethen and Bau [35]. These days it seems that we are converging to the definition suggested in that book, the spirit of which is also reflected in our own words, on the very first page of this book: "Numerical analysis is the study of algorithms for the problems of continuous mathematics."

- As mentioned earlier, in practice one often treats roundoff errors simply by using stable algorithms (no unreasonably large error accumulation) and by using double precision by default. This tends to ensure in many cases (though not all!) that these accumulated errors remain at a tolerable level. However, such a practical solution 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.

  One 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.

  Naturally, such an approach is expensive and 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. See Overton [30] for a workout.

- A move to put (more generally) **complexity theory** for numerical algorithms on firmer foundations was initiated by S. Smale and others in the 1980's. Although we are not aware of anything that arose from this volume of efforts which actually affects practical work in scientific computing, it has mushroomed into an entire organization called Foundations of Computational Mathematics (FOCM). This organization is involved with various interesting activities which concentrate on the more mathematically rich aspects of numerical computation.