Optional reading about floating point arithmetic and floating point errors

This is an excerpt from Chapter 1 of

Scientific Computing: An Introductory Survey (2nd edition)
by Michael T. Heath

Chapter 1

Scientific Computing

1.1 Introduction

The subject of this book is traditionally called *numerical analysis*. Numerical analysis is concerned with the design and analysis of algorithms for solving mathematical problems that arise in computational science and engineering. For this reason, numerical analysis has more recently become known as *scientific computing*. Numerical analysis is distinguished from most other parts of computer science in that it deals with quantities that are *continuous*, as opposed to discrete. It is concerned with functions and equations whose underlying variables—time, distance, velocity, temperature, density, pressure, stress, and the like—are continuous in nature.

Most of the problems of continuous mathematics (for example, almost any problem involving derivatives, integrals, or nonlinearities) cannot be solved, even in principle, in a finite number of steps and thus must be solved by a (theoretically infinite) iterative process that ultimately converges to a solution. In practice, of course, one does not iterate forever, but only until the answer is approximately correct, "close enough" to the desired result for practical purposes. Thus, one of the most important aspects of scientific computing is finding rapidly convergent iterative algorithms and assessing the accuracy of the resulting approximation. If convergence is sufficiently rapid, even some of the problems that *can* be solved by finite algorithms, such as systems of linear algebraic equations, may in some cases be better solved by iterative methods, as we will see.

Consequently, a second factor that distinguishes numerical analysis is its concern with approximations and their effects. Many solution techniques involve a whole series of approximations of various types. Even the arithmetic used is only approximate, for digital computers cannot represent all real numbers exactly. In addition to having the usual properties of good algorithms, such as efficiency, numerical algorithms should also be as reliable and accurate as possible despite the various approximations made along the way.

1.1.1 General Strategy

In seeking a solution to a given computational problem, a basic general strategy, which occurs throughout this book, is to replace a difficult problem with an easier one that has the same solution, or at least a closely related solution. Examples of this approach include

- Replacing infinite processes with finite processes, such as replacing integrals or infinite series with finite sums, or derivatives with finite difference quotients
- Replacing general matrices with matrices having a simpler form
- Replacing complicated functions with simple functions, such as polynomials
- Replacing nonlinear problems with linear problems
- Replacing differential equations with algebraic equations
- Replacing high-order systems with low-order systems
- Replacing infinite-dimensional spaces with finite-dimensional spaces

For example, to solve a system of nonlinear differential equations, we might first replace it with a system of nonlinear algebraic equations, then replace the nonlinear algebraic system with a linear algebraic system, then replace the matrix of the linear system with one of a special form for which the solution is easy to compute. At each step of this process, we would need to verify that the solution is unchanged, or is at least within some required tolerance of the true solution.

To make this general strategy work for solving a given problem, we must have

- An alternative problem, or class of problems, that is easier to solve
- A transformation of the given problem into a problem of this alternative type that preserves the solution in some sense

Thus, much of our effort will go into identifying suitable problem classes with simple solutions and solution-preserving transformations into those classes.

Ideally, the solution to the transformed problem is identical to that of the original problem, but this is not always possible. In the latter case the solution may only approximate that of the original problem, but the accuracy can usually be made arbitrarily good at the expense of additional work and storage. Thus, primary concerns are estimating the accuracy of such an approximate solution and establishing convergence to the true solution in the limit.

1.2 Approximations in Scientific Computation

1.2.1 Sources of Approximation

There are many sources of approximation or inexactness in computational science. Some of these occur even before computation begins:

- **Modeling:** Some physical features of the problem or system under study may be simplified or omitted (e.g., friction, viscosity).
- **Empirical measurements:** Laboratory instruments have finite precision. Their accuracy may be further limited by small sample size, or readings obtained may be subject to

random noise or systematic bias. For example, even the most careful measurements of important physical constants, such as Newton's gravitational constant or Planck's constant, typically yield values with at most eight or nine significant decimal digits.
- **Previous computations:** Input data may have been produced by a previous step whose results were only approximate.

The approximations just listed are usually beyond our control, but they still play an important role in determining the accuracy that should be expected from a computation. We will focus most of our attention on approximations over which we do have some influence. These systematic approximations that occur *during* computation include

- **Truncation or discretization:** Some features of a mathematical model may be omitted or simplified (e.g., replacing a derivative by a difference quotient or using only a finite number of terms in an infinite series).
- **Rounding** The computer representation of real numbers and arithmetic operations upon them is generally inexact.

The accuracy of the final results of a computation may reflect a combination of any or all of these approximations, and the resulting perturbations may be amplified or magnified by the nature of the problem being solved or the algorithm being used, or both. The study of the effects of such approximations on the accuracy and stability of numerical algorithms is traditionally called *error analysis*.

Example 1.1 Approximations. The surface area of the Earth might be computed using the formula

$$A = 4\pi r^2$$

for the surface area of a sphere of radius r. The use of this formula for the computation involves a number of approximations:

- The Earth is modeled as a sphere, which is an idealization of its true shape.
- The value for the radius, $r \approx 6370$ km, is based on a combination of empirical measurements and previous computations.
- The value for π is given by an infinite limiting process, which must be truncated at some point.
- The numerical values for the input data, as well as the results of the arithmetic operations performed on them, are rounded in a computer.

The accuracy of the computed result depends on all of these approximations.

1.2.2 Data Error and Computational Error

As we have just seen, some errors can be attributed to the input data, whereas others are due to subsequent computational processes. Although this distinction is not always clear-cut (rounding, for example, may affect both the input data and subsequent computational

results), it is nevertheless helpful in understanding the overall effects of approximations in numerical computations.

A typical problem can be viewed as the computation of the value of a function, say $f\colon \mathbb{R} \to \mathbb{R}$ (most realistic problems are multidimensional, but for now we consider only one dimension for illustration). Denote the true value of the input data by x, so that the desired true result is $f(x)$. Suppose that we must work with inexact input, say $\hat{x}$, and we can compute only an approximation to the function, say $\hat{f}$. Then

$$\begin{array}{rcccl} \text{Total error} & = & \hat{f}(\hat{x}) - f(x) & & \\ & = & (\hat{f}(\hat{x}) - f(\hat{x})) & + & (f(\hat{x}) - f(x)) \\ & = & \text{computational error} & + & \text{propagated data error.} \end{array}$$

The first term in the sum is the difference between the exact and approximate functions for the *same* input and hence can be considered pure *computational error*. The second term is the difference between exact function values due to error in the input and thus can be viewed as pure propagated *data error*. Note that the choice of algorithm has no effect on the propagated data error.

1.2.3 Truncation Error and Rounding Error

Similarly, computational error (that is, error made *during* the computation) can be subdivided into truncation (or discretization) error and rounding error:

- *Truncation error* is the difference between the true result (for the actual input) and the result that would be produced by a given algorithm using exact arithmetic. It is due to approximations such as truncating an infinite series, replacing a derivative by a finite difference quotient, replacing an arbitrary function by a polynomial, or terminating an iterative sequence before convergence.
- *Rounding error* is the difference between the result produced by a given algorithm using exact arithmetic and the result produced by the same algorithm using finite-precision, rounded arithmetic. It is due to inexactness in the representation of real numbers and arithmetic operations upon them, which we will consider in detail in Section 1.3.

By definition, then, computational error is simply the sum of truncation error and rounding error.

Although truncation error and rounding error can both play an important role in a given computation, one or the other is usually the dominant factor in the overall computational error. Roughly speaking, rounding error tends to dominate in purely algebraic problems with finite solution algorithms, whereas truncation error tends to dominate in problems involving integrals, derivatives, or nonlinearities, which often require a theoretically infinite solution process.

The distinctions we have made among the different types of errors are important for understanding the behavior of numerical algorithms and the factors affecting their accuracy, but it is usually not necessary, or even possible, to quantify precisely the individual types of errors. Indeed, as we will soon see, it is often advantageous to lump all of the errors together and attribute them to error in the input data.

1.2.4 Absolute Error and Relative Error

The significance of an error is obviously related to the magnitude of the quantity being measured or computed. For example, an error of 1 is much less significant in counting the population of the Earth than in counting the occupants of a phone booth. This motivates the concepts of *absolute error* and *relative error*, which are defined as follows:

$$\begin{aligned} \text{Absolute error} &= \text{approximate value} - \text{true value}, \\ \text{Relative error} &= \frac{\text{absolute error}}{\text{true value}}. \end{aligned}$$

Some authors define absolute error to be the absolute value of the foregoing difference, but we will take the absolute value explicitly when only the magnitude of the error is needed.

Relative error can also be expressed as a percentage, which is simply the relative error times 100. Thus, for example, an absolute error of 0.1 relative to a true value of 10 would be a relative error of 0.01, or 1 percent. A completely erroneous approximation would correspond to a relative error of at least 1, or at least 100 percent, meaning that the absolute error is as large as the true value. One interpretation of relative error is that if a quantity $\hat{x}$ has a relative error of about 10^{-t}, the decimal representation of $\hat{x}$ has about t correct significant digits.

Another useful way to express the relationship between absolute and relative error is the following:

$$\text{Approximate value} = (\text{true value}) \times (1 + \text{relative error}).$$

Of course, we do not usually know the true value; if we did, we would not need to bother with approximating it. Thus, we will usually merely *estimate* or *bound* the error rather than compute it exactly, because the true value is unknown. For this same reason, relative error is often taken to be relative to the approximate value rather than to the true value, as in the foregoing definition.

1.2.5 Sensitivity and Conditioning

Difficulties in solving a problem accurately are not always due to an ill-conceived formula or algorithm, but may be inherent in the problem being solved. Even with exact computation, the solution to the problem may be highly sensitive to perturbations in the input data.

A problem is said to be *insensitive*, or *well-conditioned*, if a given relative change in the input data causes a reasonably commensurate relative change in the solution. A problem is said to be *sensitive*, or *ill-conditioned*, if the relative change in the solution can be much larger than that in the input data.

More formally, we define the *condition number* of a problem f at x as

$$\text{Cond} = \frac{|\text{relative change in solution}|}{|\text{relative change in input data}|} = \frac{|(f(\hat{x}) - f(x))/f(x)|}{|(\hat{x} - x)/x|},$$

where $\hat{x}$ is a point near x. A problem is sensitive, or ill-conditioned, if its condition number is much larger than 1. Anyone who has felt a shower go from freezing to scalding, or vice

versa, at the slightest touch of the temperature control has had first-hand experience with a sensitive system.

Example 1.2 Evaluating a Function. Consider the propagated data error when a function f is evaluated for an approximate input argument $\hat{x} = x + h$ instead of the "true" input value x. We know from calculus that

$$\text{Absolute error} = f(x+h) - f(x) \approx hf'(x),$$

so that

$$\text{Relative error} = \frac{f(x+h) - f(x)}{f(x)} \approx h\frac{f'(x)}{f(x)},$$

and hence

$$\text{Cond} \approx \left|\frac{hf'(x)/f(x)}{h/x}\right| = \left|x\frac{f'(x)}{f(x)}\right|.$$

Thus, the relative error in the function value can be much larger or smaller than that in the input, depending on the properties of the function involved and the particular value of the input. For example, if $f(x) = e^x$, then the absolute error $\approx he^x$, relative error $\approx h$, and cond $\approx |x|$.

Example 1.3 Sensitivity. Consider the problem of computing values of the cosine function for arguments near $\pi/2$. Let $x \approx \pi/2$ and let h be a small perturbation to x. Then the error in computing $\cos(x+h)$ is given by

$$\text{Absolute error} = \cos(x+h) - \cos(x) \approx -h\sin(x) \approx -h,$$

and hence

$$\text{Relative error} \approx -h\tan(x) \approx \infty.$$

Thus, small changes in x near $\pi/2$ cause large relative changes in $\cos(x)$ regardless of the method for computing it. For example,

$$\cos(1.57079) = 0.63267949 \times 10^{-5},$$

whereas

$$\cos(1.57078) = 1.63267949 \times 10^{-5},$$

so that the relative change in the output, 1.58, is about a quarter of a million times larger than the relative change in the input, 6.37×10^{-6}.

1.2.6 Backward Error Analysis

Analyzing the forward propagation of errors in a computation is often very difficult. Moreover, the worst-case assumptions made at each stage often lead to a very pessimistic bound on the overall error. An alternative approach is *backward error analysis*: Consider the approximate solution obtained to be the exact solution for a modified problem, then ask how

large a modification to the original problem is required to give the result actually obtained. In other words, how much data error in the initial input would be required to explain *all* of the error in the final computed result? In terms of backward error analysis, an approximate solution to a given problem is good if it is the exact solution to a "nearby" problem.

These relationships are illustrated schematically (and not to scale) in Fig. 1.1, where x and f denote the exact input and function, respectively, $\hat{f}$ denotes the approximate function actually computed, and $\hat{x}$ denotes an input value for which the exact function would give this computed result. Note that the equality $f(\hat{x}) = \hat{f}(x)$ is due to the choice of $\hat{x}$; indeed, this requirement *defines* $\hat{x}$.

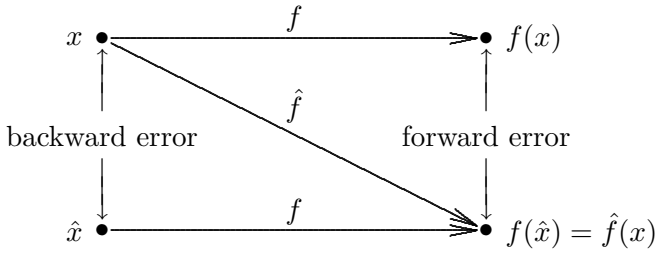

Figure 1.1: Schematic diagram of backward error analysis.

Example 1.4 Backward Error Analysis. Suppose we want a simple function for approximating the exponential function $f(x) = e^x$, and we want to examine its accuracy for the argument $x = 1$. We know that the exponential function is given by the infinite series

$$f(x) = e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots,$$

so we might consider truncating the series after, say, four terms to get the approximation

$$\hat{f}(x) = 1 + x + \frac{x^2}{2} + \frac{x^3}{6}.$$

The forward error in this approximation is then given by

$$\hat{f}(x) - f(x).$$

To determine the backward error, we must find the input value $\hat{x}$ for f that gives the output value we actually obtained for $\hat{f}$, that is, for which $f(\hat{x}) = \hat{f}(x)$. For the exponential function, we know that this value is given by

$$\hat{x} = \log(\hat{f}(x)).$$

Thus, for the particular input value $x = 1$, we have, to seven decimal places,

$$f(x) = 2.718282, \quad \hat{f}(x) = 2.666667,$$

$$\hat{x} = \log(2.666667) = 0.980829,$$

$$\text{Forward error} = \hat{f}(x) - f(x) = -0.051615,$$

$$\text{Backward error} = \hat{x} - x = -0.019171.$$

The point here is not to compare the numerical values of the forward and backward errors quantitatively, but merely to illustrate the concepts involved and to show that both are legitimate approaches to assessing accuracy. In this case, the forward error indicates that the accuracy is fairly good because the output is close to what we wanted to compute, whereas the backward error indicates that the accuracy is fairly good because the output we obtained is correct for an input that is only slightly perturbed.

1.2.7 Stability and Accuracy

The concept of stability of a computational algorithm is analogous to conditioning of a mathematical problem. Both concepts have to do with sensitivity to perturbations, but the term *stability* is usually used for algorithms and *conditioning* for problems (although stability is sometimes used for problems as well, especially in differential equations). An algorithm is *stable* if the result it produces is relatively insensitive to perturbations resulting from approximations made during the computation. From the viewpoint of backward error analysis, an algorithm is stable if the result it produces is the exact solution to a nearby problem.

Accuracy, on the other hand, refers to the closeness of a computed solution to the true solution of the problem under consideration. Stability of an algorithm does not by itself guarantee that the computed solution is accurate: accuracy depends on the conditioning of the problem as well as the stability of the algorithm. Stability tells us that the solution obtained is exact for a nearby problem, but the solution to that nearby problem is not necessarily close to the solution to the original problem unless the problem is well-conditioned. Thus, inaccuracy can result from applying a stable algorithm to an ill-conditioned problem as well as from applying an unstable algorithm to a well-conditioned problem.

1.3 Computer Arithmetic

As noted earlier, one type of approximation inevitably made in scientific computing is in representing real numbers on a computer. In this section we will examine in some detail the finite-precision arithmetic systems that are used for most scientific computations on digital computers.

1.3.1 Floating-Point Numbers

In a digital computer, the real number system of mathematics is represented approximately by a *floating-point* number system. The basic idea resembles *scientific notation*, in which a number of very large or very small magnitude is expressed as a number of moderate size times an appropriate power of ten. For example, 2347 and 0.0007396 are written as 2.347×10^3 and 7.396×10^{-4}, respectively. In this format, the decimal point moves, or *floats*, as the power of 10 changes. Formally, a floating-point number system is characterized by four integers:

β	Base or radix
t	Precision
$[L, U]$	Exponent range

By definition, any number x in the floating-point system is represented as follows:

$$x = \pm(d_0 + \frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \cdots + \frac{d_{t-1}}{\beta^{t-1}})\beta^e,$$

where

$$0 \leq d_i \leq \beta - 1, \quad i = 0, \ldots, t-1,$$

and

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

The part in parentheses, represented by the string of base-β digits $d_0 d_1 \cdots d_{t-1}$, is called the *mantissa* or *significand*, and e is called the *exponent* or *characteristic* of the floating-point number x. The portion $d_1 d_2 \cdots d_{t-1}$ of the mantissa is called the *fraction*. In a computer, the sign, exponent, and mantissa are stored in separate *fields* of a given floating-point word, each of which has a fixed width. The number zero is represented uniquely by having both its mantissa and its exponent equal to zero.

Most computers today use binary ($\beta = 2$) arithmetic, but other bases have also been used in the past, such as hexadecimal ($\beta = 16$) in IBM mainframes and $\beta = 3$ in an ill-fated Russian computer. Octal ($\beta = 8$) and hexadecimal notations are also commonly used as a convenient shorthand for writing binary numbers in groups of three or four binary digits (*bits*), respectively. For obvious reasons, decimal ($\beta = 10$) arithmetic is popular in hand-held calculators. To facilitate human interaction, a computer usually converts numerical values from decimal notation on input and to decimal notation for output, regardless of the base it uses internally. Parameters for some typical floating-point systems are given in Table 1.1, which illustrates the trade-off between precision and exponent range implied by their respective field widths. For example, working with the same 64-bit word length, the Cray system has a wider exponent range than does IEEE double precision, but at the expense of carrying less precision.

Table 1.1: Parameters for some typical floating-point systems

System	β	t	L	U
IEEE SP	2	24	-126	127
IEEE DP	2	53	$-1,022$	$1,023$
Cray	2	48	$-16,383$	$16,384$
HP calculator	10	12	-499	499
IBM mainframe	16	6	-64	63

The IEEE standard single-precision (SP) and double-precision (DP) binary floating-point systems are by far the most important today. They have been almost universally adopted for personal computers and workstations, and also for many mainframes and supercomputers as well. The IEEE standard was carefully crafted to eliminate the many anomalies and ambiguities in earlier vendor-specific floating-point implementations and has

greatly facilitated the development of portable and reliable numerical software. It also allows for sensible and consistent handling of exceptional situations, such as division by zero.

1.3.2 Normalization

A floating-point system is said to be *normalized* if the leading digit d_0 is always nonzero unless the number represented is zero. Thus, in a normalized floating-point system, the mantissa m of a given nonzero floating-point number always satisfies

$$1 \leq m < \beta.$$

(An alternative convention is that d_0 is *always* zero, in which case a floating-point number is said to be normalized if $d_1 \neq 0$, and $\beta^{-1} \leq m < 1$ instead.) Floating-point systems are usually normalized because

- The representation of each number is then unique.
- No digits are wasted on leading zeros, thereby maximizing precision.
- In a binary ($\beta = 2$) system, the leading bit is always 1 and thus need not be stored, thereby gaining one extra bit of precision for a given field width.

1.3.3 Properties of Floating-Point Systems

A floating-point number system is finite and discrete. The number of normalized floating-point numbers is

$$2(\beta - 1)\beta^{t-1}(U - L + 1) + 1$$

because there are two choices of sign, $\beta - 1$ choices for the leading digit of the mantissa, β choices for each of the remaining $t - 1$ digits of the mantissa, and $U - L + 1$ possible values for the exponent. The 1 is added because the number could be zero.

There is a smallest positive normalized floating-point number,

$$\text{Underflow level} = \text{UFL} = \beta^L,$$

which has a 1 as the leading digit and 0 for the remaining digits of the mantissa, and the smallest possible value for the exponent. There is a largest floating-point number,

$$\text{Overflow level} = \text{OFL} = \beta^{U+1}(1 - \beta^{-t}),$$

which has $\beta - 1$ as the value for each digit of the mantissa and the largest possible value for the exponent. Any number larger than OFL cannot be represented in the given floating-point system, nor can any positive number smaller than UFL.

Floating-point numbers are not uniformly distributed throughout their range, but are equally spaced only between successive powers of β. Not all real numbers are exactly representable in a floating-point system. Real numbers that are exactly representable in a given floating-point system are sometimes called *machine numbers*.

Example 1.5 Floating-Point System. An example floating-point system is illustrated

in Fig. 1.2, where the tick marks indicate all of the 25 floating-point numbers in a system having $\beta = 2$, $t = 3$, $L = -1$, and $U = 1$. For this system, the largest number is OFL $= (1.11)_2 \times 2^1 = (3.5)_{10}$, and the smallest positive normalized number is UFL $= (1.00)_2 \times 2^{-1} = (0.5)_{10}$. This is a very tiny, toy system for illustrative purposes only, but it is in fact characteristic of floating-point systems in general: at a sufficiently high level of magnification, every normalized floating-point system looks essentially like this one—grainy and unequally spaced.

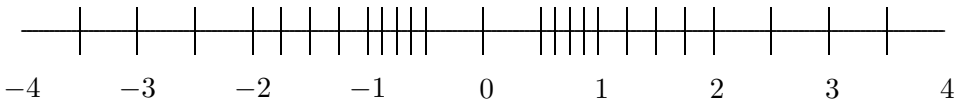

Figure 1.2: Example of a floating-point number system.

1.3.4 Rounding

If a given real number x is not exactly representable as a floating-point number, then it must be approximated by some "nearby" floating-point number. We denote the floating-point approximation of a given real number x by $\text{fl}(x)$. The process of choosing a nearby floating-point number $\text{fl}(x)$ to approximate a given real number x is called *rounding*, and the error introduced by such an approximation is called *rounding error*, or *roundoff error*. Two of the most commonly used rounding rules are

- *Chop*: The base-β expansion of x is truncated after the $(t-1)$st digit. Since $\text{fl}(x)$ is the next floating-point number towards zero from x, this rule is also sometimes called *round toward zero*.
- *Round to nearest*: $\text{fl}(x)$ is the nearest floating-point number to x; in case of a tie, we use the floating-point number whose last stored digit is even. Because of the latter property, this rule is also sometimes called *round to even*.

Rounding to nearest is the most accurate, but it is somewhat more expensive to implement correctly. Some systems in the past have used rounding rules that are cheaper to implement, such as chopping, but rounding to nearest is the default rounding rule in IEEE standard systems.

Example 1.6 Rounding Rules. Rounding the following decimal numbers to two digits using each of the rounding rules gives the following results

Number	Chop	Round to nearest	Number	Chop	Round to nearest
1.649	1.6	1.6	1.749	1.7	1.7
1.650	1.6	1.6	1.750	1.7	1.8
1.651	1.6	1.7	1.751	1.7	1.8
1.699	1.6	1.7	1.799	1.7	1.8

A potential source of additional error that is often overlooked is in the decimal-to-binary and binary-to-decimal conversions that usually take place upon input and output of floating-point numbers. Such conversions are not covered by the IEEE standard, which governs only internal arithmetic operations. Correctly rounded input and output can be obtained at reasonable cost, but not all computer systems do so. Efficient, portable routines for correctly rounded binary-to-decimal and decimal-to-binary conversions—`dtoa` and `strtod`, respectively—are available from `netlib` (see Section 1.4.1).

1.3.5 Machine Precision

The accuracy of a floating-point system can be characterized by a quantity variously known as the *unit roundoff*, *machine precision*, or *machine epsilon*. Its value, which we denote by ϵ_{mach}, depends on the particular rounding rule used. With rounding by chopping,

$$\epsilon_{\text{mach}} = \beta^{1-t},$$

whereas with rounding to nearest,

$$\epsilon_{\text{mach}} = \tfrac{1}{2}\beta^{1-t}.$$

The unit roundoff is important because it determines the maximum possible *relative error* in representing a nonzero real number x in a floating-point system:

$$\left|\frac{\text{fl}(x) - x}{x}\right| \leq \epsilon_{\text{mach}}.$$

An alternative characterization of the unit roundoff that you may sometimes see is that it is the smallest number ϵ such that

$$\text{fl}(1 + \epsilon) > 1,$$

but this is not quite equivalent to the previous definition if the round-to-even rule is used. Another definition sometimes used is that ϵ_{mach} is the distance from 1 to the next larger floating-point number, but this may differ from either of the other definitions. Although they can differ in detail, all three definitions of ϵ_{mach} have the same basic intent as measures of the granularity of a floating-point system.

For the toy illustrative system in Example 1.5, $\epsilon_{\text{mach}} = 0.25$ with rounding by chopping, and $\epsilon_{\text{mach}} = 0.125$ with rounding to nearest. For IEEE binary floating-point systems, $\epsilon_{\text{mach}} = 2^{-24} \approx 10^{-7}$ in single precision and $\epsilon_{\text{mach}} = 2^{-53} \approx 10^{-16}$ in double precision. We thus say that the IEEE single- and double-precision floating-point systems have about 7 and 16 decimal digits of precision, respectively.

Though both are "small," the unit roundoff should not be confused with the underflow level. The unit roundoff ϵ_{mach} is determined by the number of digits in the mantissa field of a floating-point system, whereas the underflow level UFL is determined by the number of digits in the exponent field. In all practical floating-point systems,

$$0 < \text{UFL} < \epsilon_{\text{mach}} < \text{OFL}.$$

1.3.6 Subnormals and Gradual Underflow

In the toy floating-point system illustrated in Fig. 1.2, there is a noticeable gap around zero. This gap, which is present to some degree in any floating-point system, is due to normalization: the smallest possible mantissa is $1.00\ldots$, and the smallest possible exponent is L, so there are no floating-point numbers between zero and β^L. If we relax our insistence on normalization and allow leading digits to be zero (but only when the exponent is at its minimum value), then the gap around zero can be "filled in" by additional floating-point numbers. For our toy illustrative system, this relaxation gains six additional floating-point numbers, the smallest positive one of which is $(0.01)_2 \times 2^{-1} = (0.125)_{10}$, as shown in Fig. 1.3.

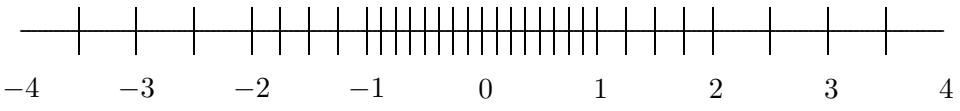

Figure 1.3: Example of a floating-point system with subnormals.

The extra numbers added to the system in this way are referred to as *subnormal* or *denormalized* floating-point numbers. Although they usefully extend the range of magnitudes representable, subnormal numbers have inherently lower precision than normalized numbers because they have fewer significant digits in their fractional parts. In particular, extending the range in this manner does not make the unit roundoff ϵ_{mach} any smaller.

Such an augmented floating-point system is sometimes said to exhibit *gradual underflow*, since it extends the lower range of magnitudes representable rather than underflowing to zero as soon as the minimum exponent value would otherwise be exceeded. The IEEE standard provides for such subnormal numbers and gradual underflow. Gradual underflow is implemented through a special reserved value of the exponent field because the leading binary digit is not stored and hence cannot be used to indicate a denormalized number.

1.3.7 Exceptional Values

The IEEE floating-point standard provides two additional special values that indicate exceptional situations:

- `Inf`, which stands for "infinity," results from dividing a finite number by zero, such as $1/0$.
- `NaN`, which stands for "not a number," results from undefined or indeterminate operations such as $0/0$, $0 * \texttt{Inf}$, or $\texttt{Inf}/\texttt{Inf}$.

`Inf` and `NaN` are implemented in IEEE arithmetic through special reserved values of the exponent field.

Whether `Inf` and `NaN` are supported at the user level in a given computing environment depends on the language, compiler, and run-time system. If available, these quantities can be helpful in designing software that deals gracefully with exceptional situations rather than

abruptly aborting the program. In MATLAB (see Section 1.4.2), for example, if Inf and NaN arise, they are propagated sensibly through a computation (e.g., 1 + Inf = Inf). It is still desirable, however, to avoid such exceptional situations entirely, if possible. In addition to alerting the user to arithmetic exceptions, these special values can also be useful as flags that cannot be confused with any legitimate numeric value. For example, NaN might be used to indicate a portion of an array that has not yet been defined.

1.3.8 Floating-Point Arithmetic

In adding or subtracting two floating-point numbers, their exponents must match before their mantissas can be added or subtracted. If they do not match initially, then the mantissa of one of the numbers must be shifted until the exponents do match. In performing such a shift, some of the trailing digits of the smaller (in magnitude) number will be shifted off the end of the mantissa field, and thus the correct result of the arithmetic operation cannot be represented exactly in the floating-point system. Indeed, if the difference in magnitude is too great, then the entire mantissa of the smaller number may be shifted completely beyond the field width so that the result is simply the larger of the operands. Another way of saying this is that if the true sum of two t-digit numbers contains more than t digits, then the excess digits will be lost when the result is rounded to t digits, and in the worst case the operand of smaller magnitude may be lost completely.

Multiplication of two floating-point numbers does not require that their exponents match—the exponents are simply summed and the mantissas multiplied. However, the product of two t-digit mantissas will in general contain up to $2t$ digits, and thus once again the correct result cannot be represented exactly in the floating-point system and must be rounded.

Example 1.7 Floating-Point Arithmetic. Consider a floating-point system with $\beta = 10$ and $t = 6$. If $x = 1.92403 \times 10^2$ and $y = 6.35782 \times 10^{-1}$, then floating-point addition gives the result $x + y = 1.93039 \times 10^2$, assuming rounding to nearest. Note that the last two digits of y have no effect on the result. With an even smaller exponent, y could have had no effect at all on the result. Similarly, floating-point multiplication gives the result $x * y = 1.22326 \times 10^2$, which discards half of the digits of the true product.

Division of two floating-point numbers may also give a result that cannot be represented exactly. For example, 1 and 10 are both exactly representable as binary floating-point numbers, but their quotient, 1/10, has a nonterminating binary expansion and thus is not a binary floating-point number.

In each of the cases just cited, the result of a floating-point arithmetic operation may differ from the result that would be given by the corresponding real arithmetic operation on the same operands because there is insufficient precision to represent the correct real result. The real result may also be unrepresentable because its exponent is beyond the range available in the floating-point system (overflow or underflow). Overflow is usually a more serious problem than underflow in the sense that there is *no* good approximation in a floating-point system to arbitrarily large numbers, whereas zero is often a reasonable approximation for arbitrarily small numbers. For this reason, on many computer systems

the occurrence of an overflow aborts the program with a fatal error, but an underflow may be silently set to zero without disrupting execution.

Example 1.8 Summing a Series. As an illustration of these issues, the infinite series

$$\sum_{n=1}^{\infty} \frac{1}{n}$$

has a finite sum in floating-point arithmetic even though the real series is divergent. At first blush, one might think that this result occurs because $1/n$ will eventually underflow, or the partial sum will eventually overflow, as indeed they must. But before either of these occurs, the partial sum ceases to change once $1/n$ becomes negligible relative to the partial sum, i.e., when $1/n < \epsilon_{\text{mach}} \sum_{k=1}^{n-1}(1/k)$, and thus the sum is finite (see Computer Problem 1.8).

As we have noted, a real arithmetic operation on two floating-point numbers does not necessarily result in another floating-point number. If a number that is not exactly representable as a floating-point number is entered into the computer or is produced by a subsequent arithmetic operation, then it must be rounded (using one of the rounding rules given earlier) to obtain a floating-point number. Because floating-point numbers are not equally spaced, the absolute error made in such an approximation is not uniform, but the relative error is bounded by the unit roundoff ϵ_{mach}.

Ideally, $x \,\texttt{flop}\, y = \text{fl}(x \,\texttt{op}\, y)$ (i.e., floating-point arithmetic operations produce correctly rounded results); and many computers, such as those meeting the IEEE floating-point standard, achieve this ideal as long as $x \,\texttt{op}\, y$ is within the range of the floating-point system. Nevertheless, some familiar laws of real arithmetic are not necessarily valid in a floating-point system. In particular, floating-point addition and multiplication are commutative but *not* associative. For example, if ϵ is a positive floating-point number slightly smaller than the unit roundoff ϵ_{mach}, then $(1 + \epsilon) + \epsilon = 1$, but $1 + (\epsilon + \epsilon) > 1$.

The failure of floating-point arithmetic to satisfy the normal laws of real arithmetic is one reason that forward error analysis can be difficult. One advantage of backward error analysis is that it permits the use of real arithmetic in the analysis.

1.3.9 Cancellation

Rounding is not the only necessary evil in finite-precision arithmetic. Subtraction between two t-digit numbers having the same sign and similar magnitudes yields a result with *fewer* than t significant digits, and hence it is always exactly representable (provided the two numbers involved do not differ in magnitude by more than a factor of two). The reason is that the leading digits of the two numbers cancel (i.e., their difference is zero). For example, again taking $\beta = 10$ and $t = 6$, if $x = 1.92403 \times 10^2$ and $z = 1.92275 \times 10^2$, then we obtain the result $x - z = 1.28000 \times 10^{-1}$, which, with only three significant digits, is exactly representable.

Despite the exactness of the result, however, such cancellation nevertheless often implies a serious loss of information. The problem is that the operands are often uncertain, owing to rounding or other previous errors, in which case the relative uncertainty in the difference

may be large. In effect, if two nearly equal numbers are accurate only to within rounding error, then taking their difference leaves only rounding error as a result.

As a simple example, if ϵ is a positive number slightly smaller than the unit roundoff ϵ_{mach}, then $(1+\epsilon)-(1-\epsilon) = 1-1 = 0$ in floating-point arithmetic, which is correct for the actual operands of the final subtraction, but the true result of the overall computation, 2ϵ, has been completely lost. The subtraction itself is not at fault: it merely signals the loss of information that had already occurred.

Of course, the loss of information is not always complete, but the fact remains that the digits lost to cancellation are the most significant, leading digits, whereas the digits lost in rounding are the least significant, trailing digits. Because of this effect, computing a small quantity as a difference of large quantities is generally a bad idea, for rounding error is likely to dominate the result. For example, summing an alternating series, such as

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

for $x < 0$, may give disastrous results because of catastrophic cancellation (see Computer Problem 1.9).

Example 1.9 Cancellation. Cancellation is not an issue only in computer arithmetic; it may also affect any situation in which limited precision is attainable, such as empirical measurements or laboratory experiments. For example, determining the distance from Manhattan to Staten Island by using their respective distances from Los Angeles will produce a very poor result unless the latter distances are known with extraordinarily high accuracy.

As another example, for many years physicists have been trying to compute the total energy of the helium atom from first principles using Monte Carlo techniques. The accuracy of these computations is determined largely by the number of random trials used. As faster computers become available and computational techniques are refined, the attainable accuracy improves. The total energy is the sum of the kinetic energy and the potential energy, which are computed separately and have opposite signs. Thus, the total energy is computed as a difference and suffers cancellation. Table 1.2 gives a sequence of values obtained over a number of years (these data were kindly provided by Dr. Robert Panoff). During this span the computed values for the kinetic and potential energies changed by only 6 percent or less, yet the resulting estimate for the total energy changed by 144 percent. The one or two significant digits in the earlier computations were completely lost in the subsequent subtraction.

Table 1.2: Computed values for the total energy of the helium atom

Year	Kinetic	Potential	Total
1971	13.0	−14.0	−1.0
1977	12.76	−14.02	−1.26
1980	12.22	−14.35	−2.13
1985	12.28	−14.65	−2.37
1988	12.40	−14.84	−2.44

Example 1.10 Quadratic Formula. Cancellation and other numerical difficulties need not involve a long series of computations. For example, use of the standard formula for the roots of a quadratic equation is fraught with numerical pitfalls. As every schoolchild learns, the two solutions of the quadratic equation

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

are given by

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

For some values of the coefficients, naive use of this formula in floating-point arithmetic can produce overflow, underflow, or catastrophic cancellation.

For example, if the coefficients are very large or very small, then b^2 or $4ac$ may overflow or underflow. The possibility of overflow can be avoided by rescaling the coefficients, such as dividing all three coefficients by the coefficient of largest magnitude. Such a rescaling does not change the roots of the quadratic equation, but now the largest coefficient is 1 and overflow cannot occur in computing b^2 or $4ac$. Such rescaling does not eliminate the possibility of underflow, but it does prevent *needless* underflow, which could otherwise occur when all three coefficients are very small.

Cancellation between $-b$ and the square root can be avoided by computing one of the roots using the alternative formula

$$x = \frac{2c}{-b \mp \sqrt{b^2 - 4ac}},$$

which has the opposite sign pattern from that of the standard formula. But cancellation inside the square root cannot be easily avoided without using higher precision (if the discriminant is small relative to the coefficients, then the two roots are close to each other, and the problem is inherently ill-conditioned).

As an illustration, we use four-digit decimal arithmetic, with rounding to nearest, to compute the roots of the quadratic equation having coefficients $a = 0.05010$, $b = -98.78$, and $c = 5.015$. For comparison, the correct roots, rounded to ten significant digits, are

$$1971.605916 \quad \text{and} \quad 0.05077069387.$$

Computing the discriminant in four-digit arithmetic produces

$$b^2 - 4ac = 9757 - 1.005 = 9756,$$

so that

$$\sqrt{b^2 - 4ac} = 98.77.$$

The standard quadratic formula then gives the roots

$$\frac{98.78 \pm 98.77}{0.1002} = 1972 \quad \text{and} \quad 0.0998.$$

The first root is the correctly rounded four-digit result, but the other root is completely wrong, with an error of about 100 percent. The culprit is cancellation, not in the sense

that the final subtraction is wrong (indeed it is exactly correct), but in the sense that cancellation of the leading digits has left nothing remaining but previous rounding errors. The alternative quadratic formula gives the roots

$$\frac{10.03}{98.78 \mp 98.77} = 1003 \quad \text{and} \quad 0.05077.$$

Once again we have obtained one fully accurate root and one completely erroneous root, but in each case it is the opposite root from the one obtained previously. Cancellation is again the explanation, but the different sign pattern causes the opposite root to be contaminated. In general, for computing each root we should choose whichever formula avoids this cancellation, depending on the sign of b.

Example 1.11 Finite Difference Approximation. Consider the finite difference approximation to the first derivative

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

We want h to be small so that the approximation will be accurate, but if h is too small, then $\text{fl}(x+h)$ may not differ from $\text{fl}(x)$. Even if $\text{fl}(x+h) \neq \text{fl}(x)$, we might still have $\text{fl}(f(x+h)) = \text{fl}(f(x))$ if f is slowly varying. In any case, we can expect some cancellation in computing the difference $f(x+h) - f(x)$. Thus, there is a trade-off between truncation error and rounding error in choosing the size of h.

If the relative error in the function values is bounded by ϵ, then the rounding error in the approximate derivative value is bounded by $2\epsilon|f(x)|/h$. The Taylor series expansion

$$f(x+h) = f(x) + f'(x)h + f''(x)h^2/2 + \cdots$$

gives an estimate of $Mh/2$ for the truncation error, where M is a bound for $|f''(x)|$. The total error is therefore bounded by

$$\frac{2\epsilon|f(x)|}{h} + \frac{Mh}{2},$$

which is minimized when

$$h = 2\sqrt{\epsilon|f(x)|/M}.$$

If we assume that the function values are accurate to machine precision and that f and f'' have roughly the same magnitude, then we obtain the rule of thumb that it is usually best to perturb about half the digits of x by taking

$$h \approx \sqrt{\epsilon_{\text{mach}}} \cdot |x|.$$

A typical example is shown in Fig. 1.4, where the error in the finite difference approximation for a particular function is plotted as a function of the stepsize h. This computation was done in IEEE single precision with $x = 1$, and the error indeed reaches a minimum at $h \approx \sqrt{\epsilon_{\text{mach}}}$. The error increases for smaller values of h because of rounding error, and increases for larger values of h because of truncation error.

The rounding error can be reduced by working with higher-precision arithmetic. Truncation error can be reduced by using a more accurate formula, such as the centered difference approximation (see Section 8.7.1)

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

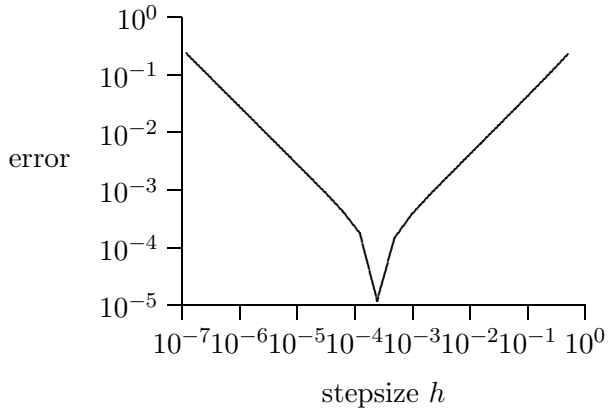

Figure 1.4: Error in finite difference approximation as a function of stepsize.

Example 1.12 Standard Deviation. The *mean* of a finite sequence of real values x_i, $i = 1, \ldots, n$, is defined by

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i,$$

and the *standard deviation* is defined by

$$\sigma = \left[\frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2 \right]^{1/2}.$$

Use of these formulas requires two passes through the data: one to compute the mean and another to compute the standard deviation. For better efficiency, it is tempting to use the mathematically equivalent formula

$$\sigma = \left[\frac{1}{n-1} \left(\sum_{i=1}^{n} x_i^2 - n\bar{x}^2 \right) \right]^{1/2}$$

to compute the standard deviation, since both the sum and the sum of squares can be computed in a single pass through the data.

Unfortunately, the single cancellation at the end of the one-pass formula is often much more damaging numerically than all of the cancellations in the two-pass formula combined. The problem is that the two quantities being subtracted in the one-pass formula are apt to

be relatively large and nearly equal, and hence the relative error in the difference may be large (indeed, the result can even be negative, causing the square root to fail).

Example 1.13 Computing Residuals. Assessing the accuracy of a computation is often difficult if one uses only the same precision as that of the computation itself. Perhaps this observation should not be surprising: if we knew the actual error, we could have used it to obtain a more accurate result in the first place.

As a simple example, suppose we are solving the scalar linear equation $ax = b$ for the unknown x, and we have obtained an approximate solution $\hat{x}$. As one measure of the quality of our answer, we wish to compute the residual $r = b - a\hat{x}$. In floating-point arithmetic,

$$a \times_{\text{fl}} \hat{x} = a\hat{x}(1 + \delta_1)$$

for some $\delta_1 \leq \epsilon_{\text{mach}}$. So

$$\begin{aligned} b -_{\text{fl}} (a \times_{\text{fl}} \hat{x}) &= [b - a\hat{x}(1 + \delta_1)](1 + \delta_2) \\ &= [r - \delta_1 a\hat{x}](1 + \delta_2) \\ &= r + \delta_2 r - \delta_1 a\hat{x} - \delta_1\delta_2 a\hat{x} \\ &\approx r + \delta_2 r - \delta_1 b. \end{aligned}$$

But $\delta_1 b$ may be as large as $\epsilon_{\text{mach}} b$, which may be as large as r. Thus, higher precision may be required to enable a meaningful computation of the residual r.

1.4 Mathematical Software

This book covers a wide range of topics in numerical analysis and scientific computing. We will discuss the essential aspects of each topic but will not have the luxury of examining any topic in great detail. To be able to solve interesting computational problems, we will often rely on mathematical software written by professionals. Leaving the algorithmic details to such software will allow us to focus on proper problem formulation and interpretation of results. We will consider only the most fundamental algorithms for each type of problem, motivated primarily by the insight to be gained into choosing an appropriate method and using it wisely. Our primary goal is to become intelligent users, rather than creators, of mathematical software.

Before citing some specific sources of good mathematical software, let us summarize the desirable characteristics that such software should possess, in no particular order of importance:

- **Reliability:** always works correctly for easy problems
- **Robustness:** usually works for hard problems, but fails gracefully and informatively when it does fail
- **Accuracy:** produces results as accurate as warranted by the problem and input data, preferably with an estimate of the accuracy achieved