Chapter 1
Computer Arithmetic and Fundamental Concepts of Computation

Abstract This chapter introduces the main concepts of error analysis used in this book. The chapter defines *reference problems* and *modified problems* and notation to distinguish them. Two kinds of modified problems are shown to be particularly important in numerical analysis, namely, *engineered* and *reverse-engineered* problems. The reader is introduced to three concepts of error: (*forward error*, *backward error*, and *residual*), to the concept of *conditioning*, and to residual-based backward error analysis—which is the method favored in this book. We define numerical properties of algorithms, including *stability* and *cost*. Finally, we apply those concepts to *floating-point arithmetic*. ◁

As we have explained in the preface, there are two main paths that one can follow with this book: a theoretical path that starts with this chapter, and a pragmatic path that starts with Chap. 4 (see Fig. 1). If you are following the theoretical path—thus reading this chapter first, before you read any other chapter—please be aware that it is among the most abstract: It provides logical and conceptual grounding for the rest of the book. We believe that the readers who prefer to consider concrete examples before encountering the general ideas of which they are instances will be better off on the first reading to start somewhere else, for example, with Chap. 4, and return to this chapter only after seeing the examples there. But if you are a theory-minded learner, then by all means this is the place to start.

1.1 Mathematical Problems and Computability of Solutions

We begin by introducing a few foundational concepts that we will use to discuss computation in the context of numerical methods, adding a few parenthetical remarks meant to contrast our perspective from others. We represent a mathematical problem by an operator φ, which has an *input* (data) space $\mathscr{I}$ as its domain and an *output* (result, solution) space $\mathscr{O}$ as its codomain:

R.M. Corless and N. Fillion, *A Graduate Introduction to Numerical Methods: From the Viewpoint of Backward Error Analysis*, DOI 10.1007/978-1-4614-8453-0_1,

$$\varphi : \mathscr{I} \to \mathscr{O} ,$$

and we write $y = \varphi(x)$. In many cases, the input and output spaces will be $\mathbb{R}^n$ or $\mathbb{C}^n$, in which case we will use the function symbols $f, g, \ldots$ and accordingly write

$$y = f(z_1, z_2, \ldots, z_n) = f(\mathbf{z}) .$$

Here, y is the (exact) solution to the problem f for the input data $\mathbf{z}$.[1] But φ need not be a function; for instance, we will study problems involving differential and integral operators. That is, in other cases, both x and y will themselves be functions.

We can delineate two general classes of computational problems related to the mathematical objects x, y, and φ:

C1. *verifying* whether a certain output y is actually the value of φ for a given input x, that is, verifying whether $y = \varphi(x)$;
C2. *finding* the output y determined by applying the map φ to a given input x, that is, finding the y such that $y = \varphi(x)$.[2]

In this classification, we consider "inverse problems," that is, trying to find an input x such that $\varphi(x)$ is a desired (known) value y, to be instances of C2 in that this corresponds to computation of the possibly many-valued inverse function $\varphi^{-1}(y)$.

The computation required by each type of problem is normally determined by an *algorithm*, that is, by a procedure performing a sequence of primitive operations leading to the solution in a finite number of steps. Numerical analysis is a mathematical reflection on the complexity and numerical properties of algorithms in contexts that involve *data error* and *computational error*.

In the study of numerical methods, as in many other branches of mathematical sciences, the reflection involves a subtle concept of *computation*. With a precise model of computation at hand, we can refine our views on what's computationally achievable, and if it turns out to be, how much effort is required.

The classical model of computation used in most textbooks on logic, computability, and algorithm analysis stems from metamathematical problems addressed in the 1930s; specifically, while trying to solve Hilbert's *Entscheidungsproblem*, Turing developed a model of primitive mathematical operations that could be performed by some type of machine affording finite but unlimited time and memory. This model, which turned out to be equivalent to other models developed independently by Gödel, Church, and others, resulted in a notion of computation based on *effective computability*. From there, we can form an idea of what is "truly feasible" by further adding constraints on time and memory.

Nonetheless, scientific computation requires an alternative, *complementary* notion of computation, because the methods and the objectives are quite different from those of metamathematics. A first important difference is the following:

[1] We use boldface font for vectors and matrices.

[2] It is normally computationally simpler to verify whether a certain value satisfies an equation than finding a value that satisfies it.

> [...] The Turing model (we call it "classical"), with its dependence on 0s and 1s, is fundamentally inadequate for giving such a foundation to the modern scientific computation, where most of the algorithms—with origins in Newton, Euler, Gauss, et al.—are *real number algorithms*. (Blum et al. 1998 3)

Blum et al. (1998) generalize the ideas found in the classical model to include operations on elements of arbitrary rings and fields. But the difference goes even deeper:

> [R]ounding errors and instability are important, and numerical analysts will always be experts in the subjects and at pains to ensure that the unwary are not tripped up by them. But our central mission is to compute quantities that are typically uncomputable, from an analytic point of view, and to do it with lightning speed. (Trefethen 1992)

Even with an improved picture of effective computability, it remains that the concept that matters for a large part of applied mathematics (including engineering) is the different idea of *mathematical tractability*, understood in a context where there are error in the data and error in computation, and where approximate answers can be entirely satisfactory. Trefethen's seemingly contradictory phrase "compute quantities that are typically uncomputable" underlines the complementarity of the two notions of computation.

This second notion of computability addresses the proper computational difficulties posed by the application of mathematics to the solution of practical problems from the outset. Certainly, both pure and applied mathematics heavily use the concepts of real and complex analysis. From real analysis, we know that every real number can be represented by a nonterminating fraction:

$$x = \lfloor x \rfloor . d_1 d_2 d_3 d_4 d_5 d_6 d_7 \cdots .$$

However, in contexts involving applications, only a finite number of digits is ever dealt with. For instance, in order to compute $\sqrt{2}$, one could use an iterative method (e.g., Newton's method, which we cover in Chap. 3) in which the number of accurate digits in the expansion will depend upon the number of iterations. A similar situation would hold if we used the first few terms of a series expansion for the evaluation of a function.

However, one must also consider another source of error due to the fact that, within each iteration (or each term), only finite-precision numbers and arithmetic operations are being used. We will find the same situation in numerical linear algebra, interpolation, numerical integration, numerical differentiation, and so forth.

Understanding the effect of limited-precision arithmetic is important in computation for problems of continuous mathematics. Since computers only store and operate on finite expressions, the arithmetic operations they process necessarily incur an error that may, in some cases, propagate and/or accumulate in alarming ways.[3] In

[3] But let's not panic: "These risks are very real, but the message was communicated all too successfully, leading to the current widespread impression that the main business of numerical analysis is coping with rounding errors (Trefethen 2008b).

this first chapter, we focus on the kind of error that arises in the context of computer arithmetic, namely, representation and arithmetic error. In fact, we will limit ourselves to the case of floating-point arithmetic, which is by far the most widely used. Thus, the two errors we will concern ourselves with are the error that results from representing a real number by a floating-point number and the error that results from computing using floating-point operations instead of real operations. For a brief review of floating-point number systems, the reader is invited to consult Appendix A.

Remark 1.1. The objective of this chapter is not so much an in-depth study of error in floating-point arithmetic as an occasion to introduce some of the most important concepts of error analysis in a context that should not pose important technical difficulty to the reader. In particular, we will introduce the concepts of residual, backward and forward error, and condition number, which will be the central concepts around which this book revolves. Together, these concepts will give solid conceptual grounds to the main theme of this book: *A good numerical method gives you nearly the right solution to nearly the right problem.* ◁

1.2 Representation and Computation Error

Floating-point arithmetic does not operate on real numbers, but rather on floating-point numbers. This generates two types of *roundoff* errors: representation error and arithmetic error. The first type of error we encounter, *representation error*, comes from the replacement of real numbers by floating-point numbers. If we let $x \in \mathbb{R}$ and $\bigcirc : \mathbb{R} \to \mathbb{F}$ be an operator for the standard rounding procedure to the nearest floating-point number[4] (see Appendix A), then the *absolute representation error* Δx is

$$\Delta x = \bigcirc x - x = \hat{x} - x \,. \tag{1.1}$$

(We will usually write $\hat{x}$ for $x + \Delta x$.) If $x \neq 0$, the *relative representation error* δx is given by

$$\delta x = \frac{\Delta x}{x} = \frac{\hat{x} - x}{x} \,. \tag{1.2}$$

From those two definitions, we obtain the following useful equality if $x \neq 0$:

$$\hat{x} = x + \Delta x = x(1 + \delta x) \,. \tag{1.3}$$

The IEEE standard described in Appendix A guarantees that $|\delta x| < \mu_M$, where μ_M is half the machine epsilon ε_M. In this book, when no specification of which IEEE

[4] In this chapter, we will always assume that x and the other real numbers are within the range of $\mathbb{F}$ for the sake of simplicity. See Appendix A for an explanation of what happens outside this domain (i.e., overflow and underflow).

standard is given, it will by default be the IEEE-754 standard described in Appendix A. In a numerical computing environment such as MATLAB, $\varepsilon_M = 2^{-52} \approx 2.2 \cdot 10^{-16}$, so that $\mu_M \approx 10^{-16}$.

The IEEE standard also guarantees that the floating-point sum of two floating-point numbers, written $\hat{z} = \hat{x} \oplus \hat{y}$,[5] is the floating-point number nearest the real sum $z = \hat{x} + \hat{y}$ of the floating-point numbers; that is, it is guaranteed that

$$\hat{x} \oplus \hat{y} = \bigcirc(\hat{x} + \hat{y}) \,. \tag{1.4}$$

In other words, the floating-point sum of two floating-point numbers is the correctly rounded real sum. As explained in Appendix A, similar guarantees are given for $\ominus, \otimes$, and $\oslash$. Paralleling the definitions of Eqs. (1.1) and (1.2), we define the absolute and relative *computation errors* (for addition) by

$$\Delta z = \hat{z} - z = (\hat{x} \oplus \hat{y}) - (\hat{x} + \hat{y}) \tag{1.5}$$

$$\delta z = \frac{\Delta z}{z} = \frac{(\hat{x} \oplus \hat{y}) - (\hat{x} + \hat{y})}{\hat{x} + \hat{y}} \,. \tag{1.6}$$

As in Eq. (1.3), we obtain

$$\hat{x} \oplus \hat{y} = \hat{z} = z + \Delta z = z(1 + \delta z) \tag{1.7}$$

with $|\delta z| < \mu_M$. Moreover, the same relations hold for multiplication, subtraction, and division. These facts give us an automatic way to transform expressions containing elementary floating-point operations into expressions containing only real quantities and operations.

Remark 1.2. Similar but not identical relationships hold for floating-point complex number operations. If $z = x + iy$, then a complex floating-point number is a pair of real floating-point numbers, and the rules of arithmetic are inherited as usual. The IEEE real floating-point guarantees discussed above translate into the following:

$$\begin{aligned} fl(z_1 \pm z_2) &= (z_1 \pm z_2)(1+\delta) & |\delta| &\le \mu_M \\ fl(z_1 z_2) &= (z_1 z_2)(1+\delta) & |\delta| &\le \sqrt{2}\gamma_2 \\ fl(z_1/z_2) &= (z_1/z_2)(1+\delta) & |\delta| &\le \sqrt{2}\gamma_7, \end{aligned} \tag{1.8}$$

where the γ_k notation [in which $\gamma_k = k\mu_M/(1 - k\mu_M)$] is as defined in Eq. (1.18) below. Division is done by a method that avoids unnecessary overflow but is slightly more complicated than the usual method (see Example 4.15). Proofs of these are given in Higham (2002). The bounds on the error are thus slightly larger for complex operations but of essentially the same character. ◁

[5] A note on notation: To make it clear that we are dealing with a floating-point counterpart of one of the elementary arithmetical operation $+, -, \times$, and $\div$, we will circle them. When we will discuss the floating-point counterparts of other operations, we will simply add "fl," such as $fl(\mathbf{x} \cdot \mathbf{y})$ for an inner product.

We can usually assume that $\sqrt{x}$ also provides the correctly rounded result, but it is not generally the case for other operations, such as e^x, $\ln x$, and the trigonometric functions (see Muller et al. 2009).

To understand floating-point arithmetic better, it is important to verify whether the standard axioms of fields are satisfied, or at least nearly satisfied. As it turns out, many standard axioms do not hold, not even nearly, and neither do their more direct consequences. Consider the following statements (for $\hat{x}, \hat{y}, \hat{z} \in \mathbb{F}$), *which are not always true in floating-point arithmetic*:

1. Associative law of $\oplus$:

$$\hat{x} \oplus (\hat{y} \oplus \hat{z}) = (\hat{x} \oplus \hat{y}) \oplus \hat{z} \tag{1.9}$$

2. Associative law of $\otimes$:

$$\hat{x} \otimes (\hat{y} \otimes \hat{z}) = (\hat{x} \otimes \hat{y}) \otimes \hat{z} \tag{1.10}$$

3. Cancellation law (for $\hat{x} \neq 0$):

$$\hat{x} \otimes \hat{y} = \hat{x} \otimes \hat{z} \;\Rightarrow\; \hat{y} = \hat{z} \tag{1.11}$$

4. Distributive law:

$$\hat{x} \otimes (\hat{y} \oplus \hat{z}) = (\hat{x} \otimes \hat{y}) \oplus (\hat{x} \otimes \hat{z}) \tag{1.12}$$

5. Multiplication cancelling division:

$$\hat{x} \otimes (\hat{y} \oslash \hat{x}) = \hat{y}. \tag{1.13}$$

In general, the associative and distributive laws fail, but commutativity still holds, as you will prove in Problem 1.15. As a result of these failures, mathematicians find it very difficult to work directly in floating-point arithmetic—its algebraic structure is weak and unfamiliar. However, thanks to the discussion above, we know how to translate a problem involving floating-point operations into a problem involving only real arithmetic on real quantities $(x, \Delta x, \delta x, \ldots)$. This approach allows us to use the mathematical structures that we are familiar with in algebra and analysis. So, instead of making our error analysis directly in floating-point arithmetic, we try to work on a problem that is *exactly* (or nearly exactly) equivalent to the original floating-point problem, by means of the study of perturbations of real (and eventually complex) quantities. This insight was first exploited systematically by J. H. Wilkinson.

1.3 Error Accumulation and Catastrophic Cancellation

In applications, it is usually the case that a large number of operations have to be done sequentially before results are obtained. In sequences of floating-point

operations, arithmetic error may accumulate. The magnitude of the accumulating error will often be negligible for well-tested algorithms.[6] Nonetheless, it is important to be aware of the possibility of massive *accumulating rounding error* in some cases. For instance, even if the IEEE standard guarantees that, for $x, y \in \mathbb{F}$, $x \oplus y = \bigcirc(x+y)$,[7] it does not guarantee that equations of the form

$$\bigoplus_{i=1}^{k} x_i = \bigcirc \sum_{i=1}^{k} x_i \,, \qquad k > 2 \tag{1.14}$$

hold true. This can potentially cause problems for the computation of sums, for instance, for the computation of an inner product $\mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^{k} x_i y_i$. In this case, the direct floating-point computation would be

$$\bigoplus_{i=1}^{k} (x_i \otimes y_i) \,, \tag{1.15}$$

summed from left to right following the indices. How big can the error be? Let us use our results from the last section in the case $n = 3$:

$$\begin{aligned} fl(\mathbf{x} \cdot \mathbf{y}) =& ((x_1 \otimes y_1) \oplus (x_2 \otimes y_2)) \oplus (x_3 \otimes y_3) \\ =& \Big(\big(x_1 y_1 (1+\delta_1) + x_2 y_2 (1+\delta_2) \big)(1+\delta_3) + x_3 y_3 (1+\delta_4) \Big)(1+\delta_5) \\ =& x_1 y_1 (1+\delta_1)(1+\delta_3)(1+\delta_5) \\ &+ x_2 y_2 (1+\delta_2)(1+\delta_3)(1+\delta_5) \\ &+ x_3 y_3 (1+\delta_4)(1+\delta_5). \end{aligned} \tag{1.16}$$

Note that the δ_is will not, in general, be identical; however, we need not pay attention to their particular values, since we are primarily interested in the fact that for real arithmetic $|\delta_i| \leq \gamma_3$ for all of them, and for complex arithmetic $|\delta_i| \leq \gamma_4$ in the θ-γ notation of Higham (2002) that we introduce below in order to clean up the presentation.

Theorem 1.1. *Consider a real floating-point system satisfying the IEEE standards, so that* $|\delta_i| < \mu_M$. *Moreover, let* $e_i = \pm 1$ *and suppose that* $n\mu_M < 1$. *Then*

$$\prod_{i=1}^{n} (1+\delta_i)^{e_i} = 1 + \theta_n \,, \tag{1.17}$$

where

[6] In fact, as explained by Higham (2002 chap. 1), errors can in some cases cancel each other out to give surprisingly accurate results.

[7] We are often only concerned with the *arithmetic* error resulting from implementing a given algorithm in floating-point arithmetic. In this case, we will drop the "$\hat{\ }$" symbol when it does not result in confusion.

$$|\theta_n| \leq \frac{n\mu_M}{1-n\mu_M} =: \gamma_n \,. \tag{1.18}$$

Notice that, for double-precision floating-point arithmetic, the supposition $n\mu_M < 1$ will almost always be satisfied. Then we can rewrite Eq. (1.16) in the real case as

$$fl(\mathbf{x}\cdot\mathbf{y}) = x_1y_1(1+\theta_3) + x_2y_2(1+\theta_3') + x_3y_3(1+\theta_2), \tag{1.19}$$

where each $|\theta_j| \leq \gamma_j$, (and where θ_3 and θ_3' each represent three different rounding errors) so that the computation error satisfies

$$|\mathbf{x}\cdot\mathbf{y} - fl(\mathbf{x}\cdot\mathbf{y})| \leq \gamma_3 \sum_{i=1}^{3} |x_iy_i| = \gamma_3 |\mathbf{x}|^T |\mathbf{y}| \,. \tag{1.20}$$

This analysis obviously generalizes to the case of n-vectors, and a similar formula can be deduced for complex vectors; as explained in the solution to (Higham 2002 Problem 3.7), all that needs to be done is to replace γ_n in the above with γ_{n+2}. However, note that this is a worst-case analysis, which returns the maximum error that can result from the mere satisfaction of the IEEE standard. In practice, it will often be much better. In fact, if you use a built-in routine for inner products, the accumulating error will be well below that (see, e.g., Problem 1.50).

Example 1.1. Another typical case in which the potential difficulty with sums poses a problem is in the computation of the value of a function using a convergent series expansion and floating-point arithmetic. Consider the simple case of the exponential function (from Forsythe 1970), $f(x) = e^x$, which can be represented by the uniformly convergent series

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots \,. \tag{1.21}$$

If we work in a floating-point system with a five-digit precision, we obtain the sum

$$\begin{aligned} e^{-5.5} &\approx 1.0000 - 5.5000 + 15.125 - 27.730 + 38.129 - 41.942 + 38.446 \\ &\quad - 30.208 + 20.768 - 12.692 + 6.9803 - 3.4902 + 1.5997 + \cdots \\ &= 0.0026363 \,. \end{aligned}$$

This is the sum of the first 25 terms, following which the first few digits do not change, perhaps leading us to believe (incorrectly) that we have reached an accurate result. But, in fact, $e^{-5.5} \approx 0.00408677$, so that $\Delta y = \hat{y} - y \approx 0.0015$. This might not seem very much, when posed in absolute terms, but it corresponds to $\delta y = 35\%$, an enormous relative error! Note, however, that it would be within what would be guaranteed by the IEEE standard for this number system. To decrease the magnitude of the maximum rounding error, we would need to add precision to the number system, thereby decreasing the magnitude of the machine epsilon. But as we will see below, this would not save us either. We are better off to use a more accurate

formula for e^{-x}, and it turns out that reciprocating the series for e^x works well for this example. See Problem 1.7. ◁

There usually are excellent built-in algorithms for the exponential function. But a similar situation could occur with the computation of values of some transcendental function for which no built-in algorithm is provided, such as the Airy function. The Airy functions (see Fig. 1.1) are solutions of the differential equation $\ddot{x} - tx = 0$

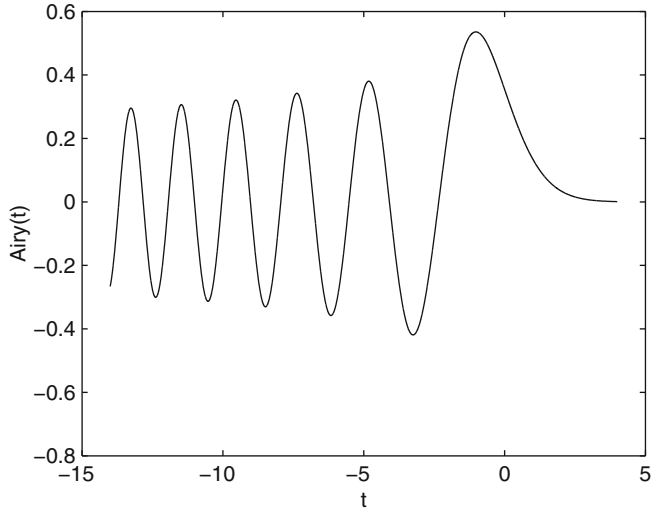


Fig. 1.1 The Airy function

with certain standard initial conditions. The first Airy function can be defined by the integral

$$\mathrm{Ai}(t) = \frac{1}{\pi}\int_0^\infty \cos\left(\frac{1}{3}\zeta^3 + t\zeta\right) d\zeta\,. \tag{1.22}$$

This function occurs often in physics. For instance, if we study the undamped motion of a weight attached to a Hookean spring that becomes linearly stiffer with time, we get the equation of motion $\ddot{x} + tx = 0$, and so the motion is described by $\mathrm{Ai}(-t)$ (Nagle et al. 2000). Similarly, the zeros of the Airy function play an important geometric role for the optics of the rainbow (Batterman 2002). And there are many more physical contexts in which it arises. So, how are we to evaluate it? The Taylor series for this function (which converges for all x) can be written as

$$\mathrm{Ai}(t) = 3^{-2/3}\sum_{n=0}^{\infty}\frac{t^{3n}}{9^n n!\Gamma(n+2/3)} - 3^{-4/3}\sum_{n=0}^{\infty}\frac{t^{3n+1}}{9^n n!\Gamma(n+4/3)} \tag{1.23}$$

(see Bender and Orszag (1978) and Chap. 3 of this book). As above, we might consider naively adding the first few terms of the Taylor series using floating-point

operations, until apparent convergence (i.e., until adding new terms does not change the solution anymore because they are too small).

Of course, true convergence would require that, for every $\varepsilon > 0$, there existed an N such that $\left|\sum_{k\geq N+1}^{M} a_k\right| < \varepsilon$ for any $M > N$, that is, that the sequence of partial sums was a Cauchy sequence. There are many tests for convergence. Indeed, for this Taylor series, we can easily use the Lagrange form of the remainder and an accurate plot of the 31st derivative of the Airy function on this interval to establish that 30 terms in the series has an error less than 10^{-16} on the interval $-12 \leq z \leq 4$. Such analysis is not always easy, though, and it is often tempting to let the machine decide when to quit adding terms; and if the terms omitted could make no difference in floating-point, then we may as well stop anyway. Of course, examples exist where this approach fails, and some of them are explored in the exercises, but when the convergence is rapid enough, as it is for this example, then this device should be harmless though a bit inefficient.

We implement this in MATLAB in the routine below:

```
function [ Ai ] = AiTaylor( z )
%AiTaylor. Try to use (naively) the explicitly-known Taylor
% series about z=0 to evaluate Ai(z). Ignore rounding errors,
% overflow/underflow, NaN. The input z may be a vector of
% complex numbers.
%
%   y = AiTaylor( z );
%
THREETWOTH  = 3.0^(-2/3);
THREEFOURTH = 3.0^(-4/3);

Ai = zeros(size(z));
zsq = z.*z;
n = 0;
zpow = ones(size(z));  % zpow = z^(3n)

term = THREETWOTH*ones(size(z))/gamma(2/3);
% recall n! = gamma(n+1)
nxtAi = Ai + term;

% Convergence is deemed to occur when adding new terms makes no
    difference numerically.
while any( nxtAi ~= Ai ),
    Ai = nxtAi;
    zpow = zpow.*z;  % zpow = z^(3n+1)
    term = THREEFOURTH*zpow/9^n/factorial(n)/gamma(n+4/3);
    nxtAi = Ai - term;
    if all( nxtAi == Ai ), break, end;
    Ai = nxtAi;
    n = n + 1;
    zpow = zpow.*zsq;  % zpow = z^(3n)
    term = THREETWOTH*zpow/9^n/factorial(n)/gamma(n+2/3);
    nxtAi = Ai + term;
end

    % We are done.  If the loop exits, Ai = AiTaylor(z).
```

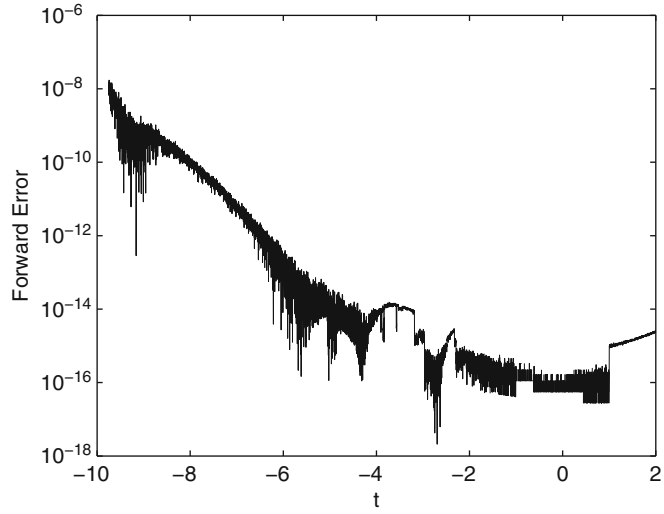


Fig. 1.2 Error in a naive MATLAB implementation of the Taylor series computation of Ai

```
36 end
```

Using this algorithm, can one expect to have a high accuracy, with error close to ε_M? Figure 1.2 displays the difference between the correct result (as computed with MATLAB's function `airy`) and the naive Taylor series approach. So, suppose we want to use this algorithm to compute $f(-12.82)$, a value near the 10th zero (counting from the origin toward $-\infty$); the absolute error is

$$\Delta y = |\mathrm{Ai}(x) - \mathrm{AiTaylor}(x)| = 0.002593213070374\,, \tag{1.24}$$

resulting in a relative error $\delta y \approx 0.277$. The solution is only accurate to two digits! Even though the series converges for all x, it is of little practical use. We examine this example in more detail in Chap. 2 when discussing the evaluation of polynomial functions.

The underlying phenomenon in the former examples, sometimes known as "the hump phenomenon," could also occur in a floating-point number system with higher precision. What happened exactly? If we consider the magnitude of some of the terms in the sum, we find out that they are much larger than the returned value (and the real value). We observe that this series is an alternating series in which the terms of large magnitude mostly cancel each other out. When such a phenomenon occurs—a phenomenon that Lehmer coined *catastrophic cancellation*—we are more likely to encounter erratic solutions. After all, how can we expect that numbers such as 38.129, a number with only five significant figures, could be used to accurately obtain the sixth or seventh figure in the answer? This explains why one must be careful in cases involving catastrophic cancellation.

Another famous example of catastrophic cancellation involves finding the roots of a degree-2 polynomial ax^2+bx+c using the quadratic equation (Forsythe 1966):

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

If we take an example for which $b^2 \gg 4ac$, catastrophic cancellation can occur. Consider this example:

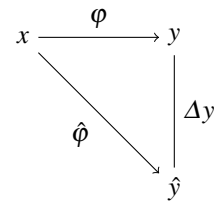
$$a = 1 \cdot 10^{-2} \qquad b = 1 \cdot 10^{7} \qquad c = 1 \cdot 10^{-2}.$$

Such numbers could easily arise in practice. Now, a MATLAB computation returns $x_{+}^{*} = 0$, which is obviously not a root of the polynomial. In this case, the answer returned is 100% wrong, in relative terms. Further exploration of this example will be made in Problem 1.18.

1.4 Perspectives on Error Analysis: Forward, Backward, and Residual-Based

The problematic cases can provoke a feeling of insecurity. When are the results provided by actual computation satisfactory? Sometimes, it is quite difficult to know intuitively whether it is the case. And how exactly should satisfaction be understood and measured? Here, we provide the concepts that will warrant confidence or non-confidence in some results based on an error analysis of the computational processes involved.

Our starting point is that problems arising in scientific computation are such that we typically do not compute the exact value $y = \varphi(x)$, for the *reference problem* φ, but instead some other more convenient value $\hat{y}$. The value $\hat{y}$ is not an exact solution of the reference problem, so that many authors regard it as an approximate solution, that is, $\hat{y} \approx \varphi(x)$. However, we will regard the quantity $\hat{y}$ as the *exact* solution of a modified problem, that is, $\hat{y} = \hat{\varphi}(x)$, where $\hat{\varphi}$ denotes the modified problem. For reasons that will become clearer later, we also call *some* modified problems *engineered problems*, because they arise on deliberately modifying φ in a way that makes computation easier or at least possible. We thus get this general picture:



(1.25)

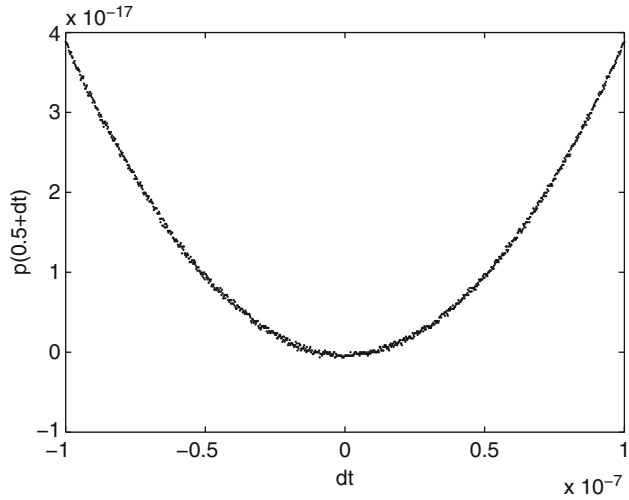


Fig. 1.3 Zooming in near a polynomial that we expect to have a double zero at $z = 1/2$, we see the curve getting "fuzzy" as we get closer because of computational error in the evaluation of the polynomial

Example 1.2. Let us consider a simple case. If we have a simple problem of addition of real numbers to do, instead of computing $y = f(x_1, x_2) = x_1 + x_2$, we might compute $\hat{y} = \hat{f}(\hat{x}_1, \hat{x}_2) = \hat{x}_1 \oplus \hat{x}_2$. Here, we regard the computation of the floating-point sum as an engineered problem. In this case, we have

$$\begin{aligned} \hat{y} &= \hat{x}_1 \oplus \hat{x}_2 = x_1(1+\delta x_1) \oplus x_2(1+\delta x_2) \\ &= \big(x_1(1+\delta x_1) + x_2(1+\delta x_2)\big)(1+\delta x_3) \\ &= (x_1 + x_2)\left(1 + \frac{x_1 \delta x_1 + x_2 \delta x_2}{x_1 + x_2}\right)(1+\delta x_3), \end{aligned} \tag{1.26}$$

and so we regard $\hat{y}$ as the exact computation of the modified formula (1.26). ◁

Similarly, if the problem is to find the zeros of a polynomial, we can use various methods that will give us so-called pseudozeros, which are usually not zeros. Instead of regarding the pseudozeros as approximate solutions of the reference problem "find the zeros," we regard those pseudozeros as the *exact* solution to the modified problem "find some zeros of nearby polynomials," which is what we mean by pseudozeros (see Chap. 2). We point out that evaluation near multiple zeros is especially sensitive to computational error; see Figs. 1.3 and 1.4.

If the problem is to find a vector $\mathbf{x}$ such that $\mathbf{A}\mathbf{x} = \mathbf{b}$, given a matrix $\mathbf{A}$ and a vector $\mathbf{b}$, we can use various methods that will give us a vector that almost satisfies the equation, but not quite. Then we can regard this vector as the solution for a matrix with slightly modified entries (see Chap. 4). The whole book is about cases of this sort arising from all branches of mathematics.

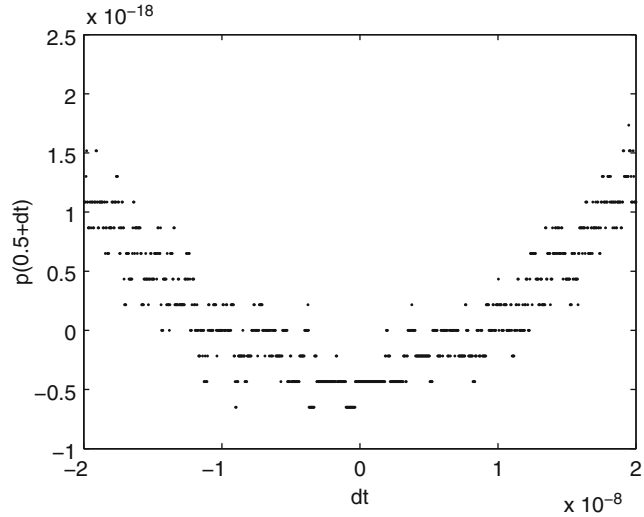


Fig. 1.4 Zooming in even closer, we see the curve broken up into discrete samples because of representation error of the computed values of the polynomial. It has also become apparent that the double zero has split to become two nearby simple zeros, each about $\sqrt{\mu_M}$ away from the reference zero $z = 1/2$. Exactly which simple zeros best represent the zeros of "the" computational polynomial is not clear-cut

What is so fruitful about this seemingly trivial change in the way the problems and solutions are discussed? Once this change of perspective is adopted, we do not focus so much on the question, "How far is the computed solution from the exact one?" (i.e., in diagram 1.25, how big is Δy?), but rather on the question, "How closely related are the original problem and the engineered problem?" (i.e., in diagram 1.25, how closely related are φ and $\hat{\varphi}$?). If the modified problem behaves closely like the reference problem, we will say it is a *nearby problem*.

The quantity labeled Δy in diagram 1.25 is called the *forward error*, which is defined by

$$\Delta y = y - \hat{y} = \varphi(x) - \hat{\varphi}(x)\,. \tag{1.27}$$

We can, of course, also introduce the *relative* forward error by dividing by y, provided $y \neq 0$. In certain contexts, the forward error is in some sense the key quantity that we want to control when designing algorithms to solve a problem. Then, a very important task is to carry a forward error analysis; the task of such an analysis is to put an upper bound on $\|\Delta y\| = \|\varphi(x) - \hat{\varphi}(x)\|$. However, as we will see, there are also many contexts in which the control of the forward error is not so crucial.

Even in contexts requiring a control of the forward error, direct forward error analysis will play a very limited role in our analyses, for a very simple reason. We engineer problems and algorithms because we don't know or don't have efficient means of computing the solution of the reference problem. But directly computing the forward error involves solving a computational problem of type C2 (as defined on p. 8), which is often unrealistic. As a result, scientific computation presents us

situations in which we usually don't know or don't have efficient ways of computing the forward error. Somehow, we need a more manageable concept that will also reveal if our computed solutions are good. Fortunately, there's another type of a priori error analysis—that is, antecedent to actual computation—one can carry out, namely, *backward error analysis*. We explain the perspective it provides in the next subsection. Then, in Sects. 1.4.2 and 1.4.3, we show how to supplement a backward error analysis with the notions of condition and residual in order to obtain an informative assessment of the forward error. Finally, in the next section, we will provide definitions for the stability of algorithms in these terms.

1.4.1 Backward Error Analysis

Let us generalize our concept of error to include any type of error, whether it comes from data error, measurement error, rounding error, truncation error, discretization error, and so forth. In effect, the success of backward error analysis comes from the fact that it treats all types of errors (physical, experimental, representational, and computational) on an equal footing. Thus, $\hat{x}$ will be some approximation of x, and Δx will be some absolute error that may be or may not be the rounding error. Similarly, in what follows, δx will be the relative error, that may or may not be the relative rounding error. The error terms will accordingly be understood as *perturbations of the initially specified data*. So, in a backward error analysis, if we consider the problem $y = \varphi(x)$, we will in general consider all the values of the data $\hat{x} = x(1+\delta x)$ satisfying a condition $|\delta x| < \varepsilon$, for some ε prescribed by the modeling context,[8] and not only the rounding errors determined by the real number x and the floating-point system. In effect, this change of perspective shifts our interest from particular values of the input data to sets of input data satisfying certain inequalities.

Now, if we consider diagram 1.25 again, we could ask: Can we find a perturbation of x that would have effects on φ comparable to the effect of changing the reference problem φ by the engineered problem $\hat{\varphi}$? Formally, we are asking: Can we find a Δx such that $\varphi(x+\Delta x) = \hat{\varphi}(x)$? The smallest such Δx is what is called the *backward error*. For input spaces whose elements are numbers, vectors, matrices, functions, and the like, we use norms as usual to determine what Δx is the backward error.[9] For other types of mixed inputs, we might have to use a set of norms for each component of the input. In case the reader needs it, Appendix C reviews basic facts about norms. The resulting general picture is illustrated in Fig. 1.5 (see, e.g., Higham 2002), and we see that this analysis amounts to *reflecting* the forward error *back* into the backward error. In effect, the question that is central to backward error analysis is, *when we modified the reference problem φ to get the engineered problem $\hat{\varphi}$, for what set of data have we actually solved the problem φ?* If solving the problem $\hat{\varphi}(x)$ amounts to having solved the problem $\varphi(x+\Delta x)$ for a Δx smaller

[8] Note that, since modeling contexts usually include the proper choice of scale, the value of ε will usually be given in relative rather than absolute terms.

[9] The choice of norm may be a delicate issue, but we will leave it aside for the moment.

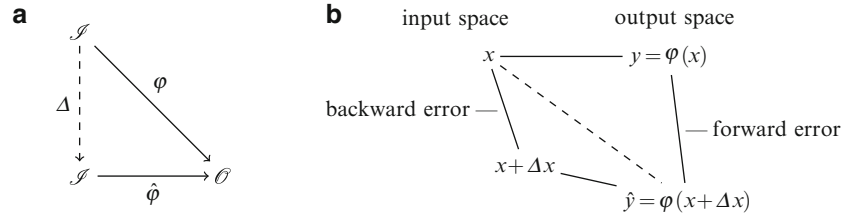


Fig. 1.5 Backward error analysis: the general picture. (**a**) Reflecting back the backward error: finding maps Δ. (**b**) Input and output space in a backward error analysis

than the perturbations inherent in the modeling context, then our solution $\hat{y}$ must be considered completely satisfactory.[10]

Adopting this approach, we benefit from the possibility of using well-known perturbation methods to talk about different problems and functions:

> The effects of errors in the data are generally easier to understand than the effects of rounding errors committed during a computation, because data errors can be analyzed using perturbation theory for the problem at hand, while intermediate rounding errors require an analysis specific to the given method. (Higham 2002 6)
>
> [T]he process of bounding the backward error of a computed solution is called *backward error analysis*, and its motivation is twofold. First, it interprets rounding errors as being equivalent to perturbations in the data. The data frequently contain uncertainties due to previous computations or errors committed in storing numbers on the computer. If the backward error is no larger than these uncertainties, then the computed solution can hardly be criticized—it may be the solution we are seeking, for all we know. The second attraction of backward error analysis is that it reduces the question of bounding or estimating the forward error to perturbation theory, which for many problems is well understood (and only to be developed once, for the given problem, and not for each method). (Higham 2002 7–8)

One can examine the effect of perturbations of the data using basic methods we know from calculus, various orders of perturbation theory, and the general methods used for the study of dynamical systems.

Example 1.3. Consider this (almost trivial!) example using only first-year calculus. Take the polynomial $p(x) = 17x^3 + 11x^2 + 2$; if there is a measurement uncertainty or a perturbation of the argument x, then how big will the effect be? One finds that

$$\Delta y = p(x+\Delta x) - p(x) = 51x^2\Delta x + 51x(\Delta x)^2 + 17(\Delta x)^3 + 22x\Delta x + 11(\Delta x)^2.$$

Now, since typically $|\Delta x| \ll 1$, we can ignore the higher degrees of Δx, so that

$$\Delta y \doteq 51x^2\Delta x\,.$$

Consequently, if $x = 1 \pm 0.1$, we get $y \doteq 35 \pm 5.1$; the perturbation in the input data has been magnified by about 50, and that would get worse if x were bigger. Also,

[10] There are cases, however, where finding such a Δx will not be possible. See Higham (2002 p. 71).

we can see from this analysis that if we want to know y to 5 decimal places, we will in general need an input accurate to 7 decimal places. ◁

Let us consider an example showing concretely how to reflect back the forward error into the backward error, in the context of floating-point arithmetic.

Example 1.4. Suppose we want to compute $y = f(x_1, x_2) = x_1^3 - x_2^3$ for the input $\mathbf{x} = [12.5, 0.333]$. For the sake of the example, suppose we have to use a computer working with a floating-point arithmetic with three-digit precision. So we will really compute $\hat{y} = ((x_1 \otimes x_1) \otimes x_1) \ominus ((x_2 \otimes x_2) \otimes x_2)$. We assume that $\mathbf{x}$ is a pair of floating-point numbers, so there is no representation error. The result of the computation is $\hat{y} = 1950$, and the exact answer is $y = 1953.014111$, leaving us with a forward error $\Delta y = 3.014111$ (or, in relative terms, $\delta y = {}^{3.014111}/_{1953.014111} \approx 1.5\%$). In a backward error analysis, we want to reflect the arithmetic (forward) error back in the data; that is, we need to find some Δx_1 and Δx_2 such that

$$\hat{y} = (12.5 + \delta x_1)^3 - (0.333 + \delta x_2)^3$$

A solution is $\Delta\mathbf{x} \approx [0.0064, 0]$ (whereby $\delta x_1 = 0.05\%$). But as one sees, the condition determines an infinite set of real solutions S, with real and complex elements. In such cases, where the entire set of solutions can be characterized, it is possible to find particular solutions, such as the solution that would minimize the 2-norm of the vector $\Delta\mathbf{x}$. See the discussions in Chaps. 4 and 6. ◁

Most of the time, we will want to use Theorem 1.1 to express the results of our backward error analyses. Consider again the case of the inner product from Eq. (1.19). The analysis we did for the three-dimensional case can be interpreted as showing that we have exactly evaluated the product $(\mathbf{x} + \Delta\mathbf{x}) \cdot \mathbf{y}$, where each perturbation is componentwise relatively small given by some θ_n (we could also have reflected back the error in $\mathbf{y}$). Specifically we have $\Delta x_1 = \theta_3 x_1$, $\Delta x_2 = \theta_3 x_2$, and $\Delta x_3 = \theta_2 x_3$. Thus, we have

$$fl(\mathbf{x} \cdot \mathbf{y}) = (\mathbf{x} + \Delta\mathbf{x}) \cdot \mathbf{y},$$

with $|\Delta\mathbf{x}| \leq \gamma_n |\mathbf{x}|$. Thus, the floating-point inner product exactly solves the reference problem for slightly perturbed data (slightly more in the case of complex data). As a result:

Theorem 1.2. *The floating-point inner product of two n-vectors is backward stable.*

Note that the order of summation does not matter for this result to obtain. However, carefully choosing the order of summation will have an impact on the forward error.

1.4.2 Condition of Problems

We have seen how we can reflect back the forward error in the backward error. Now the question we ask is: *What is the relationship between the forward and the back-*

ward error? In fact, in modeling contexts, we are not really after an expression or a value for the forward error *per se*. The only reason for which we want to estimate the forward error is to ascertain whether it is smaller than a certain user-defined "tolerance," prescribed by the modeling context. To do so, all you need is to find how the perturbations of the input data (the so-called backward error we discussed) are magnified by the reference problem. Thus, the relationship we seek lies in a problem-specific coefficient of magnification, namely, the sensitivity of the solution to perturbations in the data, which we call the *conditioning of the problem*. The conditioning of a problem is measured by the *condition number*. As for the errors, the condition number can be defined in relative and absolute terms, and it can be measured normwise or componentwise.

The *normwise relative condition number* κ_{rel} is the maximum of the ratio of the relative change in the solution to the relative change in input, which is expressed by

$$\kappa_{rel} = \sup_x \frac{\|\delta y\|}{\|\delta x\|} = \sup_x \frac{\|\Delta y/y\|}{\|\Delta x/x\|} = \sup_x \frac{\|(\varphi(\hat{x}) - \varphi(x))/\varphi(x)\|}{\|\hat{x} - x/x\|}$$

for some norm $\|\cdot\|$. As a result, we obtain the relation

$$\|\delta y\| \leq \kappa_{rel} \|\delta x\| \tag{1.28}$$

between the forward and the backward error. Knowing the backward error and the conditioning thus gives us an upper bound on the forward error.

In the same way, we can define the *normwise absolute condition number* κ_{abs} as $\sup_x \|\Delta y\|/\|\Delta x\|$, thus obtaining the relation

$$\|\Delta y\| \leq \kappa_{abs} \|\Delta x\| \,. \tag{1.29}$$

If κ has a moderate size, we say that the problem is *well-conditioned*. Otherwise, we say that the problem is *ill-conditioned*.[11] Consequently, even for a very good algorithm, the approximate solution to an ill-conditioned problem may have a large forward error.[12] It is important to observe that this fact is totally independent of any method used to compute φ. What matters is the existence of κ and what its size is.

Suppose that our problem is a scalar function. It is convenient to observe immediately that, for a sufficiently differentiable problem f, we can get an approximation of κ in terms of derivatives. Since

$$\lim_{\Delta x \to 0} \frac{\delta y}{\delta x} = \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x} \cdot \frac{x}{y} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x} \frac{x}{f(x)} = \frac{x f'(x)}{f(x)} \,,$$

the approximation of the condition number

$$\kappa_{rel} \approx \frac{|x||f'(x)|}{|f(x)|} \tag{1.30}$$

[11] When κ is unbounded, the problem is sometimes said to be *ill-posed*.

[12] Note the "may," which means that backward error analysis often provides pessimistic upper bounds on the forward error.

will provide a sufficiently good measure of the conditioning of a problem for small Δx. In the absolute case, we have $\kappa_{abs} \approx |f'(x)|$. This approximation will become useful in later chapters, and it will be one of our main tools in Chap. 3. If f is a multivariable function, the derivative $f'(x)$ will be the Jacobian matrix

$$\mathbf{J_f}(x_1, x_2, \ldots, x_n) = \left[\partial f/\partial x_1 \;\; \partial f/\partial x_2 \;\cdots\; \partial f/\partial x_n \right] ,$$

and the norm used for the computation of the condition number will be the induced matrix norm $\|\mathbf{J}\| = \max_{\|\mathbf{x}\|=1} \|\mathbf{Jx}\|$. In effect, this approximation amounts to ignoring the terms $O(\Delta x^2)$ in the Taylor expansion of $f(x+\Delta x) - f(x)$; using this approximation will thus result in a *linear error analysis*.

Though normwise condition numbers are convenient in many cases, it is often important to look at the internal structure of the arguments of the problem, for example, the dependencies between the entries of a matrix or between the components of a function vector. In such cases, it is better to use a componentwise analysis of conditioning. The relative componentwise condition number of the problem φ is the smallest number $\kappa_{rel} \geq 0$ such that

$$\max_i \frac{|f_i(\hat{x}) - f_i(x)|}{|f_i(x)|} \;\dot{\leq}\; k_{rel} \max_i \frac{|\hat{x}_i - x_i|}{|x_i|} , \quad \hat{x} \to x ,$$

where $\dot{\leq}$ indicate that the inequality holds in the limit $\Delta x \to 0$ (so, again, it holds for a linear error analysis). If the condition number is in this last form, we get a convenient theorem:

Theorem 1.3 (Deuflhard and Hohmann (2003)). *The condition number is submultiplicative; that is,*

$$\kappa_{rel}(g \circ h, x) \leq \kappa_{rel}(g, h(x)) \cdot \kappa_{rel}(h, x) .$$

In other words, the condition number of a composed problem $g \circ h$ evaluated near x is smaller than or equal to the product of the condition number of the problem h evaluated at x by the condition number of the problem g evaluated at $h(x)$. □

Consider three simple examples of condition number.

Example 1.5. Let us take the identity function $f(x) = x$ near $x = a$ (this is, of course, a trivial example). As one would expect, we get the absolute condition number

$$\kappa_{abs} = \sup \frac{|f(a+\Delta a) - f(a)|}{|\Delta a|} = \frac{|a + \Delta a - a|}{|\Delta a|} = 1 . \tag{1.31}$$

As a result, we get the relation $|\Delta y| \leq |\Delta x|$ between the forward and the backward error. κ_{abs} surely has moderate size in any context, since it does not amplify the input error. ◁

Example 1.6. Now, consider addition, $f(a,b) = a + b$. The derivative of f is

$$f'(a,b) = \left[\partial f/\partial a \quad \partial f/\partial b \right] = \left[1 \quad 1 \right] .$$

Suppose we use the 1-norm on the Jacobian matrix. Then the condition numbers are $\kappa_{abs} = \|f'(a,b)\|_1 = \| \begin{bmatrix} 1 & 1 \end{bmatrix} \|_1 = 2$ and

$$\kappa_{rel} = \frac{\left\| \begin{bmatrix} a \\ b \end{bmatrix} \right\|_1}{\|a+b\|_1} \left\| \begin{bmatrix} 1 & 1 \end{bmatrix} \right\|_1 = 2\frac{|a|+|b|}{|a+b|} . \tag{1.32}$$

(Since the function is linear, the approximation of the definitions is an equality.) Accordingly, if $|a+b| \ll |a|+|b|$, we consider the problem to be ill-conditioned. ◁

Example 1.7. Consider the problem

$$a \xrightarrow{\varphi} \{x \mid x^2 - a = 0\} ;$$

that is, evaluate x, where $x^2 - a = 0$. Take the positive root. Now here $x = \sqrt{a}$, so

$$|\delta x| = \left| \frac{f(a+\Delta a) - f(a)}{f(a)} \right| \overset{\cdot}{\leq} \left| \frac{af'(a)}{f(a)} \right| \frac{\Delta a}{a} = \frac{1}{2}\frac{\delta a}{a} \tag{1.33}$$

Thus, $\kappa = \frac{1}{2}$ is of moderate size, in a relative sense. However, note that in the absolute sense, the condition number is $\left(\sqrt{a+\Delta a} + \sqrt{a}\right)^{-1}$, which can be arbitrarily large as $a \to 0$. ◁

We will see many more examples throughout the book. Moreover, many other examples are to be found in Deuflhard and Hohmann (2003).

1.4.3 Residual-Based A Posteriori Error Analysis

The key concept we exploit in this book is the *residual*. For a given problem φ, the image y can have many forms. For example, if the reference problem φ consists in finding the roots of the equation $\xi^2 + x\xi + 2 = 0$, then for each value of x, the object y will be a set containing two numbers satisfying $\xi^2 + x\xi + 2 = 0$; that is,

$$y = \left\{ \xi \;\middle|\; \xi^2 + x\xi + 2 = 0 \right\} . \tag{1.34}$$

In general, we can then define a problem to be a map

$$x \xrightarrow{\varphi} \left\{ \xi \mid \phi(x,\xi) = 0 \right\} , \tag{1.35}$$

where $\phi(x,\xi)$ is some function of the input x and the output ξ. The function $\phi(x,\xi)$ is called the *defining function* and the equation $\phi(x,\xi) = 0$ is called the *defining equation* of the problem. On that basis, we can introduce the very important concept of *residual*: Given the reference problem φ—whose value at x is a y such that the defining equation $\phi(x,y) = 0$ is satisfied—and an engineered problem $\hat{\varphi}$, the residual r is defined by

$$r = \phi(x, \hat{y})\,. \tag{1.36}$$

As we see, we obtain the residual by substituting the computed value $\hat{y}$ (i.e., the exact solution of the engineered problem) for y as the second argument of the defining function.

Let us consider some examples in which we apply our concept of residual to various kinds of problems.

Example 1.8. The reference problem consists in finding the roots of $a_2x^2 + a_1x + a_0 = 0$. The corresponding map is $\varphi(\mathbf{a}) = \{x \,|\, \phi(\mathbf{a}, x) = 0\}$, where the defining equation is $\phi(\mathbf{a}, x) = a_2x^2 + a_1x + a_0 = 0$. Our engineered problem $\hat{\varphi}$ could consist in computing the roots to three correct places. With the resulting "pseudozeros" $\hat{x}$, we can then easily compute the residual $r = a_2\hat{x}^2 + a_1\hat{x} + a_0$. We revisit this problem in Chap. 3. ◁

Example 1.9. The reference problem consists in finding a vector $\mathbf{x}$ such that $\mathbf{Ax} = \mathbf{b}$, for a nonsingular matrix $\mathbf{A}$. The corresponding map is $\varphi(\mathbf{A}, \mathbf{b}) = \{\mathbf{x} \,|\, \phi(\mathbf{A}, \mathbf{b}, \mathbf{x}) = \mathbf{0}\}$, where the defining equation is $\phi(\mathbf{A}, \mathbf{b}, \mathbf{x}) = \mathbf{b} - \mathbf{Ax} = \mathbf{0}$. In this case, the set is a singleton since there's only one such $\mathbf{x}$. Our engineered problem could consist in using Gaussian elimination in five-digit floating-point arithmetic. With the resulting solution $\hat{\mathbf{x}}$, we can compute the residual $\mathbf{r} = \mathbf{b} - \mathbf{A}\hat{\mathbf{x}}$. We revisit this problem in Chap. 4. ◁

Example 1.10. The reference problem consists in finding a function $x(t)$ on the interval $0 < t \leq 1$ such that

$$\dot{x}(t) = f(t, x(t)) = t^2 + x(t) - \frac{1}{10}x^4(t) \tag{1.37}$$

and $x(0) = 0$. The corresponding map is

$$\varphi\big(x(0), f(t,x)\big) = \{x(t) \,|\, \phi(x(0), f(t,x), x(t)) = 0\}, \tag{1.38}$$

where the defining equation is

$$\phi\big(x(0), f(t,x), x(t)\big) = \dot{x} - f(t,x) = 0, \tag{1.39}$$

together with $x(0) = 0$ (on the given interval). In this case, if the solution exists and is unique (as happens when f is Lipschitz), the set is a singleton since there's only one such $x(t)$. Our engineered problem could consist in using, say, a continuous Runge–Kutta method. With the resulting computed solution $\hat{z}(t)$, we can compute the residual $r = \dot{\hat{z}} - f(t, \hat{z})$. We revisit this theme in Chaps. 12 and 13. ◁

Many more examples of different kinds could be included, but this should sufficiently illustrate the idea for now.

In cases similar to Example 1.10, we can rearrange the equation $r = \dot{\hat{x}} - f(t, \hat{x})$ to have $\dot{\hat{x}} = f(t, \hat{x}) + r$, so that the residual is itself a perturbation (or a backward error)

of the function defining the integral operator for our initial value problem. The new "perturbed" problem is

$$\tilde{\varphi}(x(0), f(t,x) + r(t,x)) = \{x(t) \mid \tilde{\phi}(x(0), f(t,x) + r(t,x), x(t)) = 0\}, \qquad (1.40)$$

and we observe that our computed solution $\hat{x}(t)$ is an exact solution of this problem. When such a construction is possible, we say that $\tilde{\varphi}$ is a *reverse-engineered problem*.

The remarkable usefulness of the residual comes from the fact that in scientific computation we normally choose $\hat{\varphi}$ so that we can compute it efficiently. Consequently, even if finding the solution of $\hat{\varphi}$ is a problem of type C2 (as defined on p. 8), it is normally not too computationally difficult because we engineered the problem specifically to guarantee it is so. All that remains to do to compute the residual is the evaluation of $\phi(x, \hat{y})$, a simpler problem of type C1. Thus, the computational difficulty of computing the residual is much less than that of the forward error. Accordingly, we can usually compute the residual efficiently, thereby getting a measure of the quality of our solution. Consequently, it is simpler to reverse-engineer a problem by reflecting back the residual into the backward error than by reflecting back the forward error.

Thus, the efficient computation of the residual allows us to gain important information concerning the reliability of a method on the grounds of what we have managed to compute with this method. In this context, we do not need to know as much about the intrinsic properties of a problem; we can use our computation method a posteriori to replace an a priori analysis of the reliability of the method. This allows us to use a feedback-control method to develop an adaptive procedure that controls the quality of our solution "as we go." This shows why a posteriori error estimation is tremendously advantageous in practice.

The residual-based a posteriori error analysis that we emphasize in this book thus proceeds as follows:

1. For the problem φ, use an engineered version of the problem to compute the value $\hat{y} = \hat{\varphi}(x)$.
2. Compute the residual $r = \phi(x, \hat{y})$.
3. Use the defining equation and the computed value of the residual to obtain an estimate of the backward error. In effect, this amounts to (sometimes only approximately) reflecting back the residual as a perturbation of the input data.
4. Draw conclusions about the satisfactoriness of the solution in one of two ways:
   a. If you do not require an assessment of the forward error, but only need to know that you have solved the problem for small enough perturbation Δx, conclude that your solution is satisfactory if the backward error (reflected back from the residual) is small enough.
   b. If you require an assessment of the forward error, examine the condition of the problem. If the problem is well-conditioned and the computed solution amounts to a small backward error, then conclude that your solution is satisfactory.

We still have to add some more concepts regarding the stability of algorithms, and we will do so in the next section.

But before, it is important not to mislead the reader into thinking that this type of error analysis solves *all* the problems of computational applied mathematics! There are cases involving a complex interplay of quantitative and qualitative properties that prove to be challenging. This reminds us of the following:

> A useful backward error-analysis is an explanation, not an excuse, for what may turn out to be an extremely incorrect result. The explanation seems at times merely a way to blame a bad result upon the data regardless of whether the data deserves a good result. (Kahan 2009)

Thus, even if the perspective on backward error analysis presented here is extremely fruitful, it does not cure all evils. Moreover, there are cases in which it will not even be possible to use the backward analysis framework. Here is a simple example:

Example 1.11. The outer product $\mathbf{A} = \mathbf{x}\mathbf{y}^T$ multiplies a column vector by a row vector to produce a rank-1 matrix. In floating-point arithmetic, the entries of the computed matrix $\hat{\mathbf{A}}$ will be $\hat{a}_{ij} = x_i \otimes y_j = x_i y_j(1+\delta)$ such that $|\delta| \leq \mu_M$. However, it is not possible to find perturbations $\Delta\mathbf{x}$ and $\Delta\mathbf{y}$ such that

$$\hat{\mathbf{A}} = (\mathbf{x} + \Delta\mathbf{x})(\mathbf{y} + \Delta\mathbf{y})^T .$$

See Problem 1.19. Consequently, it certainly cannot hold for small perturbations! But then, we cannot use backward error analysis to analyze this problem. ◁

1.5 Numerical Properties of Algorithms

An algorithm to solve a problem is a complete specification of how, exactly, to solve it: each step must be unambiguously defined in terms of known operations, and there must only be a finite number of steps. Algorithms to solve a problem φ correspond to the engineered problems $\hat{\varphi}$. There are many variants on the definition of an algorithm in the literature, and we will use the term loosely here. As opposed to the more restrictive definitions, we will count as algorithms methods that may fail to return the correct answer, or perhaps fail to return at all, and sometimes the method may be designed to use random numbers, thus failing to be deterministic. The key point for us is that the algorithms allow us to do computation with satisfactory results, this being understood from the point of view of mathematical tractability discussed before.

Whether $\hat{\varphi}(x)$ is satisfactory can be understood in different ways. In the literature, the algorithm-specific aspect of satisfaction is developed in terms of the numerical properties known as *numerical stability*, or just stability for short. Unfortunately "stability" is perhaps the most overused word in applied mathematics, and there is a particularly unfortunate clash with the use of the word in the theory of dynamical systems. In the terms introduced here, the concept of stability used in dynamical systems—which is a property of problems, not numerical algorithms—correspond to "well-conditioning." For algorithms, "stability" refers to the fact that an algorithm returns results that are about as accurate as the problem and the resources available allow.

Remark 1.3. The takeaway message is that, following our terminology, well-conditioning and ill-conditioning are properties of problems, while stability and instability are properties of algorithms. ◁

The first sense of numerical stability corresponds to the forward analysis point of view: an algorithm $\hat{\varphi}$ is *forward stable* if it returns a solution $y = \hat{\varphi}(x)$ with a small forward error Δy. Note that, if a problem is ill-conditioned, there will typically not be any forward stable algorithm to solve it. Nonetheless, as we explained earlier, the solution can still be satisfactory from the backward error point of view. This leads us to define *backward stability*:

Definition 1.1. An algorithm $\hat{\varphi}$ engineered to compute $y = \varphi(x)$ is *backward stable* if, for any x, there is a sufficiently small Δx such that

$$\hat{y} = f(x + \Delta x)\,, \qquad \|\Delta x\| \leq \varepsilon\,.$$

As mentioned before, what is considered "small," that is, how big ε is, is prescribed by the modeling context and, accordingly, is context-dependent. □

For example, the IEEE standard guarantees that $x \oplus y = x(1+\delta x) + y(1+\delta y)$, with $|\delta x|, |\delta y| \leq \mu_M$. Hence, the IEEE standard in effect guarantees that the algorithms for basic floating-point operations are backward stable.

Note that an algorithm returning values with large forward errors can be backward stable. This happens particularly when we are dealing with ill-conditioned problems. As Higham (2002 p. 35) puts it:

> From our algorithm we cannot expect to accomplish more than from the problem itself. Therefore we are happy when its error $\hat{f}(x) - f(x)$ lies within reasonable bounds of the error $f(\hat{x}) - f(x)$ caused by the input error.

On that basis, we can introduce the concept of stability that we will use the most. It guarantees that we obtain theoretically informative solutions, while at the same time being very convenient in practice. Often, we only establish that $\hat{y} + \Delta y = f(x + \Delta x)$ for some small Δx and Δy. We do so either for convenience of proof, or because of theoretical limitations, or because we are implementing an adaptive algorithm as we described in Sect. 1.4.3. Nonetheless, this is often sufficient from the point of view of error analysis. This leads us to the following definition (de Jong 1977; Higham 2002):

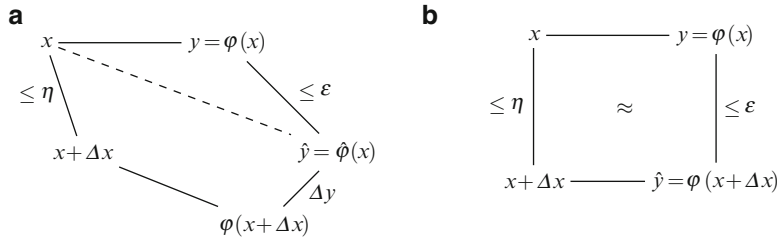


Fig. 1.6 Stability in the mixed forward–backward sense. (**a**) Representation as a commutative diagram (Higham 2002). (**b**) Representation as an "approximately" commuting diagram (Robidoux 2002). We can replace '≈' by the order to which the approximation holds

Definition 1.2. An algorithm $\hat{\varphi}$ engineered to compute $y = \varphi(x)$ is *stable in the mixed forward–backward sense* if, for any x, there are sufficiently small Δx and Δy such that

$$\hat{y} + \Delta y = f(x + \Delta x), \quad \|\Delta y\| \leq \varepsilon \|y\|, \quad \|\Delta x\| \leq \eta \|x\|. \tag{1.41}$$

See Fig. 1.6. If this case, Eq. (1.41) is interpreted as saying that $\hat{y}$ is almost the right answer for almost the right data or, alternatively, that the algorithm $\hat{\varphi}$ nearly solves the right problem for nearly the right data. □

In most cases, when we will say that an algorithm is *numerically stable* (or just stable for short), we will mean it in the mixed forward–backward sense of (1.41).

The solution to a problem $\varphi(x)$ is often obtained by replacing φ by a finite sequence of simpler problems $\varphi_1, \varphi_2, \ldots, \varphi_n$. In effect, given that the domains and codomains of the simpler subproblems match, this amount to saying that

$$\varphi(x) = \varphi_n \circ \varphi_{n-1} \circ \cdots \circ \varphi_2 \circ \varphi_1(x). \tag{1.42}$$

As we see, this is just composition of maps. For example, if the problem $\varphi(\mathbf{A}, \mathbf{b})$ is to solve the linear equation $\mathbf{Ax} = \mathbf{b}$ for $\mathbf{x}$, we might use the LU factoring (i.e., $\mathbf{A} = \mathbf{LU}$ for a lower-triangular matrix $\mathbf{L}$ and an upper-triangular matrix $\mathbf{U}$) factorization to obtain the two equations

$$\mathbf{Ly} = \mathbf{Pb} \tag{1.43}$$

$$\mathbf{Ux} = \mathbf{y}. \tag{1.44}$$

We have then decomposed $\mathbf{x} = \varphi(\mathbf{A}, \mathbf{b})$ into two problems; the first problem $\mathbf{y} = \varphi_1(\mathbf{L}, \mathbf{P}, \mathbf{b})$ consists in the simple task of solving a lower-triangular system and the second problem $\mathbf{x} = \varphi_2(\mathbf{U}, \mathbf{y})$ consists in the simple task of solving an upper-triangular system (see Chap. 4).

Remark 1.4. Such decompositions are hardly unique. A good choice of $\varphi_1, \varphi_2, \ldots, \varphi_n$ may lead to a good algorithm for solving φ in this way: Solve $\varphi_1(x)$ using its stable algorithm to get $\hat{y}_1$, then solve $\varphi_2(\hat{y}_1)$ using its stable algorithm to get $\hat{y}_2$, and so on. If the subproblems φ_1 and φ_2 are also well-conditioned, by Theorem 1.3, it follows that the resulting composed numerical algorithm for φ is numerically stable. (The same principle can be use as a very accurate rule of thumb for the formulations of the condition number not covered by Theorem 1.3). ◁

The *converse* statement is also very useful: Decomposing a *well-conditioned* φ into two *ill-conditioned* subproblems $\varphi = \varphi_2 \circ \varphi_1$ will usually result in an *unstable* algorithm for φ, even if stable algorithms are available for each of the subproblems (unless, as seems unlikely, the errors in $\hat{\varphi}_1$ and $\hat{\varphi}_2$ cancel each other out).

To a large extent, any numerical methods book is about decomposing problems into subproblems, and examining the correct numerical strategies to solve the subproblems. In fact, if you take any problem in applied mathematics, chances are that it will involve as subproblems things such as evaluating functions, finding roots of

polynomials, solving linear systems, finding eigenvalues, interpolating function values, and so on. Thus, in each chapter, a small number of "simple" problems will be examined, so that you can construct the composed algorithm that is appropriate for your own composed problems.

1.6 Complexity and Cost of Algorithms

So far, we have focused on the accuracy and stability of numerical methods. In fact, most of the content of this book will focus more on accuracy and stability than on cost of algorithms and complexity of problems. Nonetheless, we will at times need to address issues of complexity. To evaluate the cost of some method, we need two elements: (1) a count of the number of elementary operations required by its execution and (2) a measure of the amount of resources required by each type of elementary operation, or group of operations. Following the traditional approach, we will only include the first element in our discussion.[13] Thus, when we will discuss the cost of algorithms, we will really be discussing the number of floating-point operations (*flops*[14]) required for the termination of an algorithm. Moreover, following a common convention, we will consider one flop to be one addition, one multiplication, and one comparison.

Example 1.12. If we take two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$, the inner product

$$\mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^{n} x_i y_i = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n$$

requires n flops. Thus, the multiplication of two arbitrary $n \times n$ matrices requires n^3 flops, since each entry is computed by an inner product.

Note that the order of operations may affect the flop count. If we also take $\mathbf{z} \in \mathbb{R}^n$, there will be a difference between $(\mathbf{x}\mathbf{y}^T)\mathbf{z}$ and $\mathbf{x}(\mathbf{y}^T\mathbf{z})$. In the former case, the first operation is an outer product forming an $n \times n$ matrix, which require n^2 flops. It is followed by a matrix–vector multiplication; this is equivalent to n inner products, each requiring n flops. Thus, the cost is $n^2 + n^2 = 2n^2$. However, if we instead compute $\mathbf{x}(\mathbf{y}^T\mathbf{z})$, the first operation is a scalar product (n flops) and the second operation is a multiplication of a vector by a scalar (n flops), which together require $2n$ flops. ◁

Note that sometimes the vectors, matrices, or other objects on which we operate will have a particular structure that we will be able to exploit to produce more efficient algorithms. The *computational complexity* of a problem is the cost of the algorithm

[13] The second element, particularly memory resources, is very relevant in practice today; in fact, possibly *more* relevant than the cost of floating-point, since one can demonstrate that computation time can sometimes be accurately be accurately estimated from memory requirements alone.

[14] In computer science, the acronym "flops" is sometimes used to denote flop/s, or floating-point operations per second. Here, the "s" only marks the plural of "flop."

solving this problem with the least cost, that is, what it would require to solve the problem using the cheapest method.

Typically, we will not be too concerned with the exact flop count. Rather, we will only provide an order of magnitude determined by the highest-order terms of the expressions for the flop count. Thus, if an algorithm taking an input of size n requires $n^2/2+n+2$ flops, we will simply say that its cost is $n^2/2+O(n)$ flops, or even just $O(n^2)$ flops. This way of describing cost is achieved by means of *asymptotic notation*. The asymptotic notation uses the symbols Θ, O, Ω, o and ω to describe the comparative rate of growth of functions of n as n becomes large. In this book, however, we will only use the big-O and small-o notation, which are defined as follows:

$$\begin{aligned} f(n) = O(g(n)) \quad &\text{iff} \quad \exists c > 0 \exists n_0 \forall n \geq n_0 \quad \text{such that} \quad 0 \leq f(n) \leq c \cdot g(n) \\ f(n) = o(g(n)) \quad &\text{iff} \quad \forall c > 0 \exists n_0 \forall n \geq n_0 \quad \text{such that} \quad 0 \leq f(n) \leq c \cdot g(n). \end{aligned} \tag{1.45}$$

Intuitively, a function $f(n)$ is $O(g(n))$ when its rate of growth with respect to n is the same or less than the rate of growth of $g(n)$, as depicted in Fig. 1.7 (in other words, $\lim_{n\to\infty} f(n)/g(n)$ is bounded). A function $f(n)$ is $o(g(n))$ in the same circumstances, except that the rate of growth of $f(n)$ must be strictly less than $g(n)$'s (in other words, $\lim_{n\to\infty} f(n)/g(n)$ is zero). Thus, $g(n)$ is an asymptotic upper bound for $f(n)$. However, with the small-o notation, the bound is not tight.

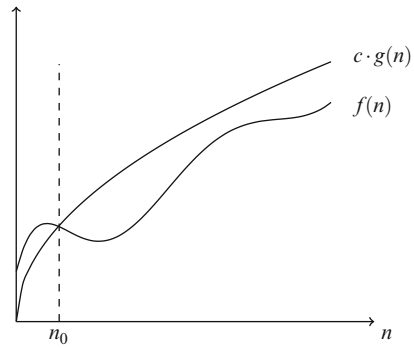


Fig. 1.7 Asymptotic notation: $f(n) = O(g(n))$ if, for some c, $cg(n)$ asymptotically bounds $f(n)$ above as $n \to \infty$

In our context, if we say that the cost of a method is $O(g(n))$, we mean that as n becomes large, the number of flops required will be at worst $g(n)$ times a constant. Some standard terminology to qualify cost growth, from smaller to larger growth rate, in introduced in Table 1.1. We will also use this notation when writing sums. See Sect. 2.8.

This notation is also used to discuss *accuracy*, and work-accuracy relationships. We will often want to analyze the cost of an algorithm as a function of a parameter, typically a dimension, say n, or a grid size, say h. The interesting limits are as the dimension goes to infinity or as the grid size goes to zero. The residual or backward error will typically go to zero as some power of h or inverse power of n (sometimes faster, in which case we say the convergence is *spectral*). If we have the error behaving as $\|\Delta\| = O(h^p)$ as $h \to 0$, we say the method has *order* p, and similarly if $\|\Delta\| = O(n^{-p})$. The asymptotic O-symbol hides a constant that may or may not be important.

Table 1.1 Common growth rates

The cost $f(n)$ is	The growth rate if the cost is
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \log n)$	Quasilinear
$O(n^2)$	Quadratic
$O(n^k)$, $k = 2, 3, \ldots$	Polynomial
$O(c^n)$	Exponential

One useful trick for *measuring* the rate of convergence of a problem is to use a Fibonacci sequence[15] of dimension parameters, measure the errors for each dimension (this is typically easy if the error is a backward error), and plot the results on a log–log graph. This is called a work-accuracy diagram because the work increases as n increases (usually as a power of n itself) and the slope of the line of best fit then estimates p. We do this at several places in the book.

1.7 Notes and References

For a presentation of the classical model of computation, see, for instance, Davis (1982), Brassard and Bratley (1996), Pour-El and Richards (1989), and for a specific discussion of what is "truly feasible," see Immerman (1999).

Brent and Zimmermann (2011) provides a recent extensive discussion of algorithms and models of computer arithmetic, including floating-point arithmetic.

For an alternative, more formal presentation of the concepts presented here to systematically articulate backward error analysis, see Deuflhard and Hohmann (2003 chap. 2). The "reflecting back" terminology goes back to Wilkinson (1963). For a good historical essay on backward error analysis, see Grcar (2011).

Many other examples of numerical surprises can be found in the paper "Numerical Monsters," by Essex et al. (2000). The experience of W. Kahan in constructing floating-point systems to minimize the impact on computation has been

[15] Why use a Fibonacci sequence or something like it? Because they grow exponentially, but not as quickly as doubling the dimension does, and this often produces a more pleasing density of results on the graph.

presented in a systematic way in the entertaining and informative talk (Kahan and Darcy 1998). Many of his other papers are available on his website at http://www.cs.berkeley.edu/~wkahan.

Problems

Theory and Practice

1.1. Suppose you're an investor who will get interest daily (for an annual rate of, say 5%) on $1,000,000. Your interest can be calculated in one of two ways: (a) The sum is calculated every day, and rounded to the nearest cent. This new amount will be used to calculate your sum on the next day. (b) Your sum is calculated only once at the end of the year with the formula $M_f = M_i(1 + i_d)^d$, and then rounded to the nearest cent.

1. Which method should you choose? How big is the difference? How much smaller is it than the worst-case scenario obtained from mere satisfaction of the IEEE standard? Explain in terms of floating-point error.
2. If the rounding procedure used for the floating-point arithmetic was "round toward zero," would you make the same decision?

Explain the correspondence between computational error and real-world operations.

1.2. An important value to determine in the analysis of alternating current circuits is the capacitive reactance X_C, which is given by

$$X_C = -\frac{1}{2\pi f C},$$

where f is the frequency of the signal (in Hertz) and C is the capacitance (in Farads). It is common to encounter the values $f = 60$ Hz while C is the range of picofarads (i.e., $10^{-12}F$). Given this, could we expect MATLAB to accurately compute the reactive capacitance in common situations? Also, look up common values for the tolerance in the value of C provided by manufacturers. Would the rounding error be smaller than the error due to the tolerance? In at most a few sentences, discuss the significance of your last answer for assessing the quality of computed solutions.

1.3. Suppose you want to use MATLAB to help you with some calculations involved in special relativity. A common quantity to compute is the Lorentz factor γ defined by

$$\gamma = \frac{1}{\sqrt{1 - \dfrac{v^2}{c^2}}},$$

where v is the relative velocity between two inertial frames in m/s and c is the speed of light, which is nearly equal to $299,792,458$ m/s. Will MATLAB provide

results sufficiently precise to identify the relativistic effect of a vehicle moving at $v = 100.000$ km/h? Given the significant figures of v, is MATLAB's numerical result satisfactory? Compare your results with what you obtain from

$$(1-x^2)^{-1/2} = 1 + x^2/2 + O(x^4)\,. \tag{1.46}$$

1.4. Computing powers z^n for integers n and floating-point z can be done by simple repeated multiplication, or by a more efficient method known as *binary powering*. If $n = 2k+1$ is odd, replace the problem with that of computing $z \cdot z^{2k}$. If $n = 2k$ is even, replace the problem with that of computing $z^k \cdot z^k$. Recursively descend until $k = 1$. This can be done efficiently by looking at the bit pattern of the original n. Estimate the maximum number of multiplications are performed.

1.5. Suppose a, b are real but not machine-representable numbers. Compare the accuracy of computing $(a+b)^2$ as written and computing instead using the expanded form $a^2 + 2ab + b^2$. Are both methods backward stable? Mixed forward–backward stable? Would the difference between the methods, if any, become more important for $(a+b)^n$, $n > 2$? Give examples supporting your theoretical conclusions. You may use Problem 1.4.

1.6. Show that, for $a \neq 0$ and $b \neq 0$,

1. $25n^3 + n^2 + n - 4 = O(n^3)$;
2. any linear function $f(n) = an + b$ is $O(n^k)$ and $o(n^k)$ for integers $k \geq 2$;
3. no quasilinear function $an\log(bn)$ is $o(n\log(n))$.

1.7. Rework Example 1.1 using five-digit precision as before but compute instead $\exp(5.5)$ and then take the reciprocal. This uses the same numbers printed in the text, just all with positive signs. Is your final answer more accurate?

1.8. Euler was the first to discover[16] that

$$\sum_{k=1}^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6}\,. \tag{1.47}$$

Write a program in MATLAB to sum the terms of this series in order (i.e., start with $k = 1$, then $k = 2$, *etc.*) until the double-precision sum is unaffected by adding another term. Record the number of terms taken (we found nearly 10^8). Compare the answer to $pi^2/6$ and record the relative accuracy. Write another program to evaluate the same sum in decreasing order of the values of k. What is the relative forward error in this case? Is it different? Is it significantly different? That is, is the accumulation of error reduced for a sum of positive numbers if we add the numbers from smallest to largest? (Higham 2002 1.12.3). Use the "integral test" from first-year calculus to estimate the true error in stopping the sum where you did, and estimate the number of terms you would have to take to get $\pi^2/6$ to as much accuracy as you could in double precision simply by summing terms.

[16] For a historical discussion of this, see the beautiful book Hairer and Wanner (1996), if you like, though it is not necessary for this problem.

1.9. The value of the Riemann zeta-function at 3 is

$$\zeta(3) = \sum_{k\geq 1} \frac{1}{k^3}\,. \tag{1.48}$$

Quite a lot is known about this number, but all you are asked to do here is to compute its value by simple summation as in the `AiTaylor` program and as in the previous problem, by simply adding terms until the next term is so small it has no effect after rounding. Use the integral test to estimate the actual error of your sum, and to estimate how many terms you would really need to sum to get double-precision accuracy. If you summed in reverse order, would you get an accurate answer?

1.10. Testing for convergence in floating-point arithmetic is tricky due to computational error. Discuss foreseeable difficulties and workarounds. In particular, you may wish to address the "method" used in the function `AiTaylor` of this chapter, namely to assume "convergence" of a series if adding a term t to a sum s produces $\hat{s} = s \oplus t$ that, after rounding, exactly equals s. Consider in particular what happens if you use this method on a *divergent* sum such as the harmonic series $H = 1 + 1/2 + 1/3 + 1/4 + \cdots$. (This is the source of many Internet arguments, by the way, but there is a clear and unambiguously correct way of looking at it.)

1.11. Show that computing the sum $\sum_{i=1}^{n} x_i$ naively term by term (a process called *recursive summation*) produces the result

$$\bigoplus_{i=1}^{n} x_i = \sum_{i=1}^{n} x_i(1+\delta_i), \tag{1.49}$$

where each $|\delta_i| \leq \gamma_{n+1-i}$ if $i \geq 2$ and $|\delta_1| \leq \gamma_{n-1}$ if $i = 1$.

There are a surprising number of different ways to sum n real numbers, as discussed in Higham (2002). Using Kahan's algorithm for *compensated summation* as described below instead returns the computed sum

$$\sum_{i=1}^{n} x_i(1+\delta_i), \tag{1.50}$$

where now each $|\delta_i| < 2\mu_M + O(n\mu_M)$, according to Higham (2002) (you do not have to prove this). That is, compensated summation gains a factor of n in backward accuracy.

The algorithm in question is the following:

Require: A vector **x** with n components.
  $s := x_1$
  $c := 0$
  for i from 2 to n **do**
    $y := x_i - c$
    $t := s + y$
    $c := (t - s) - y$ % the order is important, and the parentheses too!

$s := t$
end for
return s, the sum of the components of $\mathbf{x}$

Using some examples, compare the accuracy of naive recursive summation and of Kahan's sum. If you can, show that Eq. (1.50) really holds for your examples (Goldberg 1991).

1.12. For this problem, we work with a four-digit precision floating-point system. Note that $1+1=2$ gives no error since $1 \in \mathbb{F}$. In exact arithmetic, $1/3 + 1/3 = 2/3$, but floating-point operations imply that $\frac{1}{3}(1+\delta_1) + \frac{1}{3}(1+\delta_2) = 0.667$, from which we find that $\delta_1 + \delta_2 = (3 \cdot 0.6667 - 2) = 0.0001$. Show that $\max(|\delta_1|, |\delta_2|)$ is minimized if $|\delta_1| = |\delta_2| = 5 \cdot 10^{-5}$.

1.13. The following expressions are theoretically equivalent:

$$\begin{aligned}
s_1 &= 10^{20} + 17 - 10 + 130 - 10^{20} \\
s_2 &= 10^{20} - 10 + 130 - 10^{20} + 17 \\
s_3 &= 10^{20} + 17 - 10^{20} - 10 + 130 \\
s_4 &= 10^{20} - 10 - 10^{20} + 130 + 17 \\
s_5 &= 10^{20} - 10^{20} + 17 - 10 + 130 \\
s_6 &= 10^{20} + 17 + 130 - 10^{20} - 10.
\end{aligned}$$

Nonetheless, a standard computer returns the values $0, 17, 120, 147, 137, -10$ (see, e.g.,Kulisch 2002 [8]). These errors stem from the fact that catastrophic cancellation takes place due to very different orders of magnitude. For each expression, find some values of δx_i, $1 \le i \le 5$, such that

$$s = x_1(1+\delta x_1) + x_2(1+\delta x_2) + x_3(1+\delta x_3) + x_4(1+\delta x_4) + x_5(1+\delta x_5)$$

with $|\delta x_i| < \mu_M$. In each case, find $\min \|\delta \mathbf{x}\|$.

1.14. Show that Eqs. (1.9), (1.10), (1.11), (1.12), and (1.13) do not generally hold for floating-point numbers.

1.15. Other laws of algebra for inequalities fail in floating-point arithmetic. Let $a, b, c, d \in \mathbb{F}$ (Parhami 2000 325):

1. Show that if $a < b$, then $a \oplus c \le b \oplus c$ holds for all c; that is, adding the same value to both sides of a strict inequality cannot affect its direction but may change the strict "$<$" relationship to "$\le$."
2. Show that if $a < b$ and $c < d$, then $a \oplus c \le b \oplus d$.
3. Show that if $c > 0$ and $a < b$, then $a \otimes c \le b \otimes c$.

Assume that none of a, b, c, and d are `NaN`.

1.16. Higham (2002 1.12.2) considers what happens in floating-point computation when one first takes square roots repeatedly, and then squares the result repeatedly.

We here look at a slight variation, which (surprisingly for such an innocuous-looking computation) has something to do with an ancient but effective algorithm known as *Briggs' method* (Higham 2004 chapter 11). Here, write a MATLAB function that accepts a vector **x** as input, takes the square root 52 times, and then squares the result 52 times: theoretically achieving nothing. Call your function `Higham`. The algorithm is indicated below.

Require: A vector **x**
  for i from 1 to 52 **do**
    $x := \sqrt{x}$
  end for
  for i from 1 to 52 **do**
    $x := x^2$
  end for
  return a vector x, surprisingly different to the input

Then run

```
x = logspace( 0, 1, 2013 );
y = Higham( x );
plot( x, y, 'k.', x, x, '--' )
```

Explain the graph (see Fig. 1.8). (Hint: Identify the points where $y = x$ after all.)

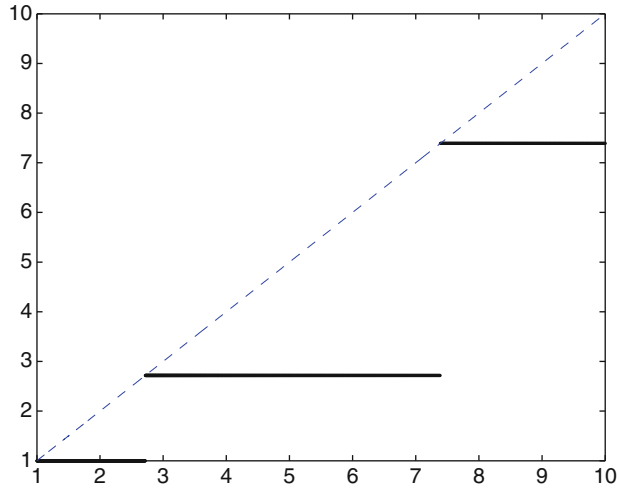


Fig. 1.8 The results of the code in Problem 1.16

1.17. We now know that unfortunate subtractions bring loss of significant figures. In fact, the subtraction per se does not introduce much error, but it reveals earlier error. On that basis, compare the following two methods to find the two roots of a second-degree polynomial:

1. Use the two cases of the quadratic formula;
2. Using the fact that $x_+x_- = c$ (where $x^2 + bx + c = 0$, i.e., $a = 1$), keep the root among the two obtained with the quadratic formula that has the largest absolute value, and find the other one using the equation $x_+x_- = c$.

Which method is more accurate? Explain.

Investigations and Projects

1.18. Consider the quadratic equation $x^2 + 2bx + 1 = 0$.

1. Show by the quadratic formula or otherwise that $x = -b \pm \sqrt{b^2 - 1}$ and that the product of the two roots is 1.
2. Plot $(-b + \sqrt{b^2 - 1})(-b - \sqrt{b^2 - 1})$, which is supposed to be 1, on a logarithmic scale in MATLAB as follows:

```
b = logspace( 6, 7.5, 1001 );
one = (-b-sqrt(b.^2-1) ).*(-b+sqrt(b.^2-1));
plot( b, one, '.' )
```

3. Using no more than one page of handwritten text (about a paragraph of typed text), partly explain why the plot looks the way it does.
4. If $b \gg 1$, which is more accurately evaluated in floating-point arithmetic, $-b - \sqrt{b^2 - 1}$ or $-b + \sqrt{b^2 - 1}$? Why?

1.19. Consider the outer product of two vectors $\mathbf{x} \in \mathbb{C}^m$ and $\mathbf{y} \in \mathbb{C}^n$: $\mathbf{P} = \mathbf{x}\mathbf{y}^H \in \mathbb{C}^{m \times n}$ with $p_{ij} = x_i \bar{y}_j$. Show that if $mn > m + n$, then rounding errors in computing this object cannot be modeled as a backward error; in other words, show that $\hat{\mathbf{P}}$ is not the exact outer product of any two perturbations $\mathbf{x} + \Delta\mathbf{x}$ and $\mathbf{y} + \Delta\mathbf{y}$.

1.20. Let $p = 1/2$. Consider the mathematically equivalent sums

$$1 = \sum_{k \geq 1} \frac{1}{k^p} - \frac{1}{(k+1)^p} \tag{1.51}$$

$$= \sum_{k \geq 1} \frac{(k+1)^p - k^p}{k^p (k+1)^p} \tag{1.52}$$

$$= \sum_{k \geq 1} \frac{1}{k^p (k+1)^p ((k+1)^p + k^p)} \,. \tag{1.53}$$

Which of these is the most accurate to evaluate in floating-point using naive recursive summation? Why?

1.21 (Zeno's paradox: The dichotomy). One of the classical paradoxes of Zeno runs (more or less) as follows: A pair of dance partners are two units apart and wish to move together, each moving one unit. But for that to happen, they must first each move half a unit. After they have done that, then they must move half of the

distance remaining. After *that*, they must move half the distance yet remaining, and so on. Since there are an infinite number of steps involved, logical difficulties seem to arise and indeed there is puzzlement in the first-year calculus class regarding things like this, although in modern models of analysis this paradox has long since been resolved. Roughly speaking, the applied mathematics view is that after a finite number of steps, the dancers are close enough for all practical purposes!

In MATLAB, we might phrase the paradox as follows. By symmetry, replace one partner with a mirror. Then start the remaining dancer off at $s_0 = 0$. The mirror is thus at $s = 1$. The first move is to $s_1 = s_0 + (1 - s_0)/2$. The second move is to $s_2 = s_1 + (1 - s_1)/2$. The third move is to $s_3 = s_2 + (1 - s_2)/2$, and so on. This suggests the following loop.

```
s = 0
i = 0
while s < 1,
    i = i+1;
    s = s + (1-s)/2;
end
disp( sprintf( 'Dancer␣reached␣the␣mirror␣in␣%d␣steps', i) )
```

Does this loop terminate? If so, how many iterations does it take?

http://www.springer.com/978-1-4614-8452-3

A Graduate Introduction to Numerical Methods
From the Viewpoint of Backward Error Analysis

Corless, R.; Fillion, N.

2013, XXXIX, 869 p. 194 illus., 10 illus. in color.,
Hardcover
ISBN: 978-1-4614-8452-3