CHAPTER 2

Numerics and Error Analysis

CONTENTS

NUMERICAL analysis introduces a shift from working with `int`s and `long`s to `float`s and `double`s. This seemingly innocent transition shatters intuition from integer arithmetic, requiring adjustment of how we must think about basic algorithmic design and implementation. Unlike discrete algorithms, numerical algorithms cannot always yield exact solutions even to well-studied and well-posed problems. Operation counting no longer reigns supreme; instead, even basic techniques require careful analysis of the trade-offs among timing, approximation error, and other considerations. In this chapter, we will explore the typical factors affecting the quality of a numerical algorithm. These factors set numerical algorithms apart from their discrete counterparts.

2.1 STORING NUMBERS WITH FRACTIONAL PARTS

Most computers store data in *binary* format. In binary, integers are decomposed into powers of two. For instance, we can convert 463 to binary using the following table:

1	1	1	0	0	1	1	1	1
2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

This table illustrates the fact that 463 has a unique decomposition into powers of two as:

$$\begin{aligned} 463 &= 256 + 128 + 64 + 8 + 4 + 2 + 1 \\ &= 2^8 + 2^7 + 2^6 + 2^3 + 2^2 + 2^1 + 2^0. \end{aligned}$$

All positive integers can be written in this form. Negative numbers also can be represented either by introducing a leading sign bit (e.g., 1 for "positive" and 0 for "negative") or by using a "two's complement" trick.

The binary system admits an extension to numbers with fractional parts by including *negative* powers of two. For instance, 463.25 can be decomposed by adding two slots:

1	1	1	0	0	1	1	1	1.	0	1
2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}

Representing fractional parts of numbers this way, however, is not nearly as well-behaved as representing integers. For instance, writing the fraction $1/3$ in binary requires infinitely many digits:

$$\frac{1}{3} = 0.0101010101\cdots_2 .$$

There exist numbers at all scales that cannot be represented using a finite binary string. In fact, all irrational numbers, like $\pi = 11.00100100001\ldots_2$, have infinitely long expansions regardless of which (integer) base you use!

Since computers have a finite amount of storage capacity, systems processing values in $\mathbb{R}$ instead of $\mathbb{Z}$ are forced to approximate or restrict values that can be processed. This leads to many points of confusion while coding, as in the following snippet of C++ code:

```
double x = 1.0;
double y = x / 3.0;
if (x == y*3.0) cout << "They are equal!";
else cout << "They are NOT equal.";
```

Contrary to intuition, this program prints `"They are NOT equal."` Why? Since $1/3$ cannot be written as a finite-length binary string, the definition of `y` makes an approximation, rounding to the nearest number representable in the `double` data type. Thus, `y*3.0` is *close to but not exactly* 1. One way to fix this issue is to allow for some tolerance:

```
double x = 1.0;
double y = x / 3.0;
if (fabs(x-y*3.0) < numeric_limits<double>::epsilon)
     cout << "They are equal!";
else cout << "They are NOT equal.";
```

Here, we check that `x` and `y*3.0` are near enough to each other to be reasonably considered identical rather than whether they are exactly equal. The tolerance `epsilon` expresses how far apart values should be before we are confident they are different. It may need to be adjusted depending on context. This example raises a crucial point:

Rarely if ever should the operator == and its equivalents be used on fractional values. Instead, some *tolerance* should be used to check if they are equal.

There is a trade-off here: the size of the tolerance defines a line between equality and "close-but-not-the-same," which must be chosen carefully for a given application.

The error generated by a numerical algorithm depends on the choice of *representations* for real numbers. Each representation has its own compromise among speed, accuracy, range of representable values, and so on. Keeping the example above and its resolution in mind, we now consider a few options for representing numbers discretely.

2.1.1 Fixed-Point Representations

The most straightforward way to store fractions is to use a *fixed* decimal point. That is, as in the example above, we represent values by storing 0-or-1 coefficients in front of powers of two that range from 2^{-k} to 2^{ℓ} for some $k, \ell \in \mathbb{Z}$. For instance, representing all nonnegative values between 0 and 127.75 in increments of $1/4$ can be accomplished by taking $k = 2$ and $\ell = 7$; in this case, we use 10 binary digits total, of which two occur after the decimal point.

The primary advantage of this representation is that many arithmetic operations can be carried out using the same machinery already in place for integers. For example, if a and b are written in fixed-point format, we can write:

$$a + b = (a \cdot 2^k + b \cdot 2^k) \cdot 2^{-k}.$$

The values $a \cdot 2^k$ and $b \cdot 2^k$ are integers, so the summation on the right-hand side is an integer operation. This observation essentially shows that fixed-point addition can be carried out using integer addition essentially by "ignoring" the decimal point. In this way, rather than needing specialized hardware, the preexisting integer arithmetic logic unit (ALU) can carry out fixed-point mathematics quickly.

Fixed-point arithmetic may be fast, but it suffers from serious precision issues. In particular, it is often the case that the output of a binary operation like multiplication or division can require more bits than the operands. For instance, suppose we include one decimal point of precision and wish to carry out the product $1/2 \cdot 1/2 = 1/4$. We write $0.1_2 \times 0.1_2 = 0.01_2$, which gets truncated to 0. More broadly, it is straightforward to combine fixed-point numbers in a reasonable way and get an unreasonable result.

Due to these drawbacks, most major programming languages do not by default include a fixed-point data type. The speed and regularity of fixed-point arithmetic, however, can be a considerable advantage for systems that favor timing over accuracy. Some lower-end graphics processing units (GPU) implement only fixed-point operations since a few decimal points of precision are sufficient for many graphical applications.

2.1.2 Floating-Point Representations

One of many numerical challenges in scientific computing is the extreme range of scales that can appear. For example, chemists deal with values anywhere between 9.11×10^{-31} (the mass of an electron in kilograms) and 6.022×10^{23} (the Avogadro constant). An operation as innocent as a change of units can cause a sudden transition between scales: The same observation written in kilograms per lightyear will look considerably different in megatons per mile. As numerical analysts, we are charged with writing software that can transition gracefully between these scales without imposing unnatural restrictions on the client.

Scientists deal with similar issues when recording experimental measurements, and their methods can motivate our formats for storing real numbers on a computer. Most prominently, one of the following representations is more compact than the other:

$$6.022 \times 10^{23} = 602,200,000,000,000,000,000,000.$$

Not only does the representation on the left avoid writing an unreasonable number of zeros, but it also reflects the fact that we may not know Avogadro's constant beyond the second 2.

In the absence of exceptional scientific equipment, the difference between 6.022×10^{23} and $6.022 \times 10^{23} + 9.11 \times 10^{-31}$ likely is negligible, in the sense that this tiny perturbation is dwarfed by the error of truncating 6.022 to three decimal points. More formally, we say that 6.022×10^{23} has only four *digits of precision* and probably represents some range of possible measurements $[6.022 \times 10^{23} - \varepsilon, 6.022 \times 10^{23} + \varepsilon]$ for some $\varepsilon \approx 0.001 \times 10^{23}$.

Our first observation allowed us to shorten the representation of 6.022×10^{23} by writing it in *scientific notation*. This number system separates the "interesting" digits of a number from its order of magnitude by writing it in the form $a \times 10^e$ for some $a \sim 1$ and $e \in \mathbb{Z}$. We call this format the *floating-point* form of a number, because unlike the fixed-point setup in

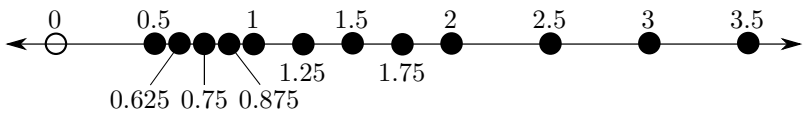

Figure 2.1 The values from Example 2.1 plotted on a number line; typical for floating-point number systems, they are unevenly spaced between the minimum (0.5) and the maximum (3.5).

§2.1.1, the decimal point "floats" so that a is on a reasonable scale. Usually a is called the *significand* and e is called the *exponent*.

Floating-point systems are defined using three parameters:

- The *base* or *radix* $b \in \mathbb{N}$. For scientific notation explained above, the base is $b = 10$; for binary systems the base is $b = 2$.
- The *precision* $p \in \mathbb{N}$ representing the number of digits used to store the significand.
- The range of exponents $[L, U]$ representing the allowable values for e.

The expansion looks like:

$$\underbrace{\pm}_{\text{sign}} \underbrace{(d_0 + d_1 \cdot b^{-1} + d_2 \cdot b^{-2} + \cdots + d_{p-1} \cdot b^{1-p})}_{\text{significand}} \times \underbrace{b^e}_{\text{exponent}},$$

where each digit d_k is in the range $[0, b-1]$ and $e \in [L, U]$. When $b = 2$, an extra bit of precision can be gained by *normalizing* floating-point values and assuming the most significant digit d_0 is one; this change, however, requires special treatment of the value 0.

Floating-point representations have a curious property that can affect software in unexpected ways: Their spacing is uneven. For example, the number of values representable between b and b^2 is the same as that between b^2 and b^3 even though usually $b^3 - b^2 > b^2 - b$. To understand the precision possible with a given number system, we will define the *machine precision* ε_m as the smallest $\varepsilon_m > 0$ such that $1 + \varepsilon_m$ is representable. Numbers like $b + \varepsilon_m$ are not expressible in the number system because ε_m is too small.

Example 2.1 (Floating-point)**.** Suppose we choose $b = 2$, $L = -1$, and $U = 1$. If we choose to use three digits of precision, we might choose to write numbers in the form

$$1.\square\square \times 2^{\square}.$$

Notice this number system does not include 0. The possible significands are $1.00_2 = 1_{10}$, $1.01_2 = 1.25_{10}$, $1.10_2 = 1.5_{10}$, and $1.11_2 = 1.75_{10}$. Since $L = -1$ and $U = 1$, these significands can be scaled by $2^{-1} = 0.5_{10}$, $2^0 = 1_{10}$, and $2^1 = 2_{10}$. With this information in hand, we can list all the possible values in our number system:

Significand	$\times 2^{-1}$	$\times 2^0$	$\times 2^1$
1.00_{10}	0.500_{10}	1.000_{10}	2.000_{10}
1.25_{10}	0.625_{10}	1.250_{10}	2.500_{10}
1.50_{10}	0.750_{10}	1.500_{10}	3.000_{10}
1.75_{10}	0.875_{10}	1.750_{10}	3.500_{10}

These values are plotted in Figure 2.1; as expected, they are unevenly spaced and bunch toward zero. Also, notice the gap between 0 and 0.5 in this sampling of values; some

number systems introduce evenly spaced *subnormal* values to fill in this gap, albeit with less precision. Machine precision for this number system is $\varepsilon_m = 0.25$, the smallest displacement possible above 1.

By far the most common format for storing floating-point numbers is provided by the IEEE 754 standard. This standard specifies several classes of floating-point numbers. For instance, a double-precision floating-point number is written in base $b = 2$ (as are all numbers in this format), with a single $\pm$ sign bit, 52 digits for d, and a range of exponents between -1022 and 1023. The standard also specifies how to store $\pm\infty$ and values like NaN, or "not-a-number," reserved for the results of computations like $^{10}/_{0}$.

IEEE 754 also includes agreed-upon conventions for rounding when an operation results in a number not represented in the standard. For instance, a common unbiased strategy for rounding computations is *round to nearest, ties to even,* which breaks equidistant ties by rounding to the nearest floating-point value with an even least-significant (rightmost) bit. There are many equally legitimate strategies for rounding; agreeing upon a single one guarantees that scientific software will work identically on all client machines regardless of their particular processor or compiler.

2.1.3 More Exotic Options

For most of this book, we will assume that fractional values are stored in floating-point format unless otherwise noted. This, however, is not to say that other numerical systems do not exist, and for specific applications an alternative choice might be necessary. We acknowledge some of those situations here.

The headache of inexact arithmetic to account for rounding errors might be unacceptable for some applications. This situation appears in computational geometry, e.g., when the difference between *nearly* and *completely* parallel lines may be a difficult distinction to make. One solution might be to use *arbitrary-precision arithmetic,* that is, to implement fractional arithmetic without rounding or error of any sort.

Arbitrary-precision arithmetic requires a specialized implementation and careful consideration for what types of values you need to represent. For instance, it might be the case that rational numbers $\mathbb{Q}$, which can be written as ratios $^{a}/_{b}$ for $a, b \in \mathbb{Z}$, are sufficient for a given application. Basic arithmetic can be carried out in $\mathbb{Q}$ without any loss in precision, as follows:

$$\frac{a}{b} \times \frac{c}{d} = \frac{ac}{bd} \qquad\qquad \frac{a}{b} \div \frac{c}{d} = \frac{ad}{bc}.$$

Arithmetic in the rationals precludes the existence of a square root operator, since values like $\sqrt{2}$ are irrational. Also, this representation is nonunique since, e.g., $^{a}/_{b} = {}^{5a}/_{5b}$, and thus certain operations may require additional routines for simplifying fractions. Even after simplifying, after a few multiplies and adds, the numerator and denominator may require many digits of storage, as in the following sum:

$$\frac{1}{100} + \frac{1}{101} + \frac{1}{102} + \frac{1}{103} + \frac{1}{104} + \frac{1}{105} = \frac{188463347}{3218688200}.$$

In other situations, it may be useful to bracket error by representing values alongside error estimates as a pair $a, \varepsilon \in \mathbb{R}$; we think of the pair (a, ε) as the range $a \pm \varepsilon$. Then, arithmetic operations also update not only the value but also the error estimate, as in

$$(x \pm \varepsilon_1) + (y \pm \varepsilon_2) = (x + y) \pm (\varepsilon_1 + \varepsilon_2 + \text{error}(x + y)),$$

where the final term represents an estimate of the error induced by adding x and y. Maintaining error bars in this fashion keeps track of confidence in a given value, which can be informative for scientific calculations.

2.2 UNDERSTANDING ERROR

With the exception of the arbitrary-precision systems described in §2.1.3, nearly every computerized representation of real numbers with fractional parts is forced to employ rounding and other approximations. Rounding, however, represents one of many sources of error typically encountered in numerical systems:

- *Rounding* or *truncation* error comes from rounding and other approximations used to deal with the fact that we can only represent a finite set of values using most computational number systems. For example, it is impossible to write π exactly as an IEEE 754 floating-point value, so in practice its value is truncated after a finite number of digits.
- *Discretization* error comes from our computerized adaptations of calculus, physics, and other aspects of continuous mathematics. For instance, a numerical system might attempt to approximate the derivative of a function $f(t)$ using *divided differences*:

  $$f'(t) \approx \frac{f(t+\varepsilon) - f(t)}{\varepsilon}$$

  for some fixed choice of $\varepsilon > 0$. This approximation is a legitimate and useful one that we will study in Chapter 14, but since we must use a finite $\varepsilon > 0$ rather than taking a limit as $\varepsilon \to 0$, the resulting value for $f'(t)$ is only accurate to some number of digits.
- *Modeling* error comes from having incomplete or inaccurate descriptions of the problems we wish to solve. For instance, a simulation predicting weather in Germany may choose to neglect the collective flapping of butterfly wings in Malaysia, although the displacement of air by these butterflies might perturb the weather patterns elsewhere. Furthermore, constants such as the speed of light or acceleration due to gravity might be provided to the system with a limited degree of accuracy.
- *Input* error can come from user-generated approximations of parameters of a given system (and from typos!). Simulation and numerical techniques can help answer "what if" questions, in which exploratory choices of input setups are chosen just to get some idea of how a system behaves. In this case, a highly accurate simulation might be a waste of computational time, since the inputs to the simulation were so rough.

Example 2.2 (Computational physics)**.** Suppose we are designing a system for simulating planets as they revolve around the sun. The system essentially solves Newton's equation $F = ma$ by integrating forces forward in time. Examples of error sources in this system might include:

- *Rounding error:* Rounding the product ma to IEEE floating-point precision
- *Discretization error:* Using divided differences as above to approximate the velocity and acceleration of each planet
- *Modeling error:* Neglecting to simulate the moon's effects on the earth's motion within the planetary system

- *Input error:* Evaluating the cost of sending garbage into space rather than risking a Wall-E style accumulation on Earth, but only guessing the total amount of garbage to jettison monthly

2.2.1 Classifying Error

Given our previous discussion, the following two numbers might be regarded as having the same amount of error:

$$1 \pm 0.01$$
$$10^5 \pm 0.01.$$

Both intervals $[1 - 0.01, 1 + 0.01]$ and $[10^5 - 0.01, 10^5 + 0.01]$ have the same width, but the latter appears to encode a more confident measurement because the error 0.01 is much smaller *relative* to 10^5 than to 1.

The distinction between these two classes of error is described by distinguishing between *absolute* error and *relative* error:

Definition 2.1 (Absolute error)**.** The *absolute error* of a measurement is the difference between the approximate value and its underlying true value.

Definition 2.2 (Relative error)**.** The *relative error* of a measurement is the absolute error divided by the true value.

Absolute error is measured in input units, while relative error is measured as a percentage.

Example 2.3 (Absolute and relative error)**.** Absolute and relative error can be used to express uncertainty in a measurement as follows:

$$\text{Absolute: } 2 \text{ in} \pm 0.02 \text{ in}$$
$$\text{Relative: } 2 \text{ in} \pm 1\%$$

Example 2.4 (Catastrophic cancellation)**.** Suppose we wish to compute the difference $d \equiv 1 - 0.99 = 0.01$. Thanks to an inaccurate representation, we may only know these two values up to ± 0.004. Assuming that we can carry out the subtraction step without error, we are left with the following expression for absolute error:

$$d = 0.01 \pm 0.008.$$

In other words, we know d is somewhere in the range $[0.002, 0.018]$. From an absolute perspective, this error may be fairly small. Suppose we attempt to calculate relative error:

$$\frac{|0.002 - 0.01|}{0.01} = \frac{|0.018 - 0.01|}{0.01} = 80\%.$$

Thus, although 1 and 0.99 are known with relatively small error, the difference has enormous relative error of 80%. This phenomenon, known as *catastrophic cancellation*, is a danger associated with subtracting two nearby values, yielding a result close to zero.

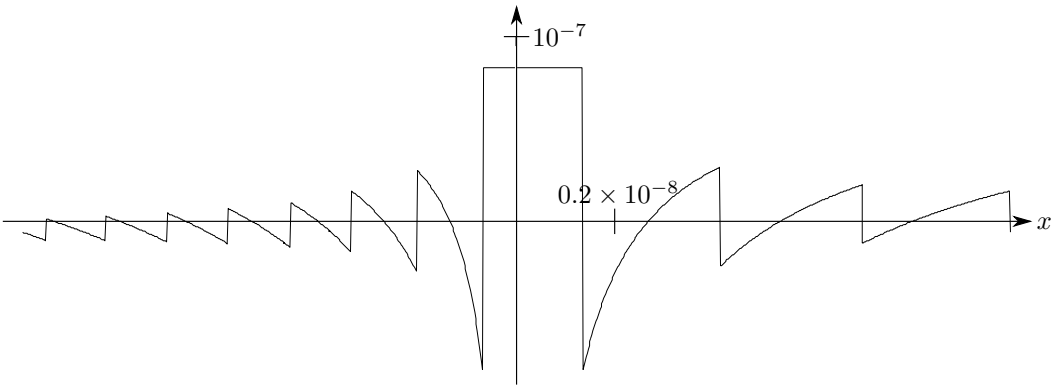

Figure 2.2 Values of $f(x)$ from Example 2.5, computed using IEEE floating-point arithmetic.

Example 2.5 (Loss of precision in practice)**.** Figure 2.2 plots the function

$$f(x) \equiv \frac{e^x - 1}{x} - 1,$$

for evenly spaced inputs $x \in [-10^{-8}, 10^{-8}]$, computed using IEEE floating-point arithmetic. The numerator and denominator approach 0 at approximately the same rate, resulting in loss of precision and vertical jumps up and down near $x = 0$. As $x \to 0$, in theory $f(x) \to 0$, and hence the relative error of these approximate values blows up.

In most applications, the *true* value is unknown; after all, if it were known, there would be no need for an approximation in the first place. Thus, it is difficult to compute relative error in closed form. One possible resolution is to be conservative when carrying out computations: At each step take the largest possible error estimate and propagate these estimates forward as necessary. Such conservative estimates are powerful in that when they are small we can be very confident in our output.

An alternative resolution is to acknowledge *what* you can measure; this resolution requires somewhat more intricate arguments but will appear as a theme in future chapters. For instance, suppose we wish to solve the equation $f(x) = 0$ for x given a function $f : \mathbb{R} \to \mathbb{R}$. Our computational system may yield some x_{est} satisfying $f(x_{\text{est}}) = \varepsilon$ for some ε with $|\varepsilon| \ll 1$. If x_0 is the true root satisfying $f(x_0) = 0$, we may not be able to evaluate the difference $|x_0 - x_{\text{est}}|$ since x_0 is unknown. On the other hand, by evaluating f we can compute $|f(x_{\text{est}}) - f(x_0)| \equiv |f(x_{\text{est}})|$ since $f(x_0) = 0$ by definition. This difference of f values gives a proxy for error that still is zero exactly when $x_{\text{est}} = x_0$.

This example illustrates the distinction between *forward* and *backward* error. Forward error is the most direct definition of error as the difference between the approximated and actual solution, but as we have discussed it is not always computable. Contrastingly, backward error is a calculable proxy for error *correlated* with forward error. We can adjust the definition and interpretation of backward error as we consider different problems, but one suitable—if vague—definition is as follows:

Definition 2.3 (Backward error)**.** The *backward error* of an approximate solution to a numerical problem is the amount by which the problem statement would have to change to make the approximate solution exact.

This definition is somewhat obtuse, so we illustrate its application to a few scenarios.

Example 2.6 (Linear systems)**.** Suppose we wish to solve the $n \times n$ linear system $A\vec{x} = \vec{b}$ for $\vec{x} \in \mathbb{R}^n$. Label the true solution as $\vec{x}_0 \equiv A^{-1}\vec{b}$. In reality, due to rounding error and other issues, our system yields a near-solution $\vec{x}_{\text{est}}$. The forward error of this approximation is the difference $\vec{x}_{\text{est}} - \vec{x}_0$; in practice, this difference is impossible to compute since we do not know $\vec{x}_0$. In reality, $\vec{x}_{\text{est}}$ is the *exact* solution to a modified system $A\vec{x} = \vec{b}_{\text{est}}$ for $\vec{b}_{\text{est}} \equiv A\vec{x}_{\text{est}}$; thus, we might measure backward error in terms of the difference $\vec{b} - \vec{b}_{\text{est}}$. Unlike the forward error, this error is easily computable without inverting A, and $\vec{x}_{\text{est}}$ is a solution to the problem exactly when backward (or forward) error is zero.

Example 2.7 (Solving equations, from [58], Example 1.5)**.** Suppose we write a function for finding square roots of positive numbers that outputs $\sqrt{2} \approx 1.4$. The forward error is $|1.4 - \sqrt{2}| \approx 0.0142$. The backward error is $|1.4^2 - 2| = 0.04$.

These examples demonstrate that backward error can be much easier to compute than forward error. For example, evaluating forward error in Example 2.6 required inverting a matrix A while evaluating backward error required only multiplication by A. Similarly, in Example 2.7, transitioning from forward error to backward error replaced square root computation with multiplication.

2.2.2 Conditioning, Stability, and Accuracy

In nearly any numerical problem, zero backward error implies zero forward error and vice versa. A piece of software designed to solve such a problem surely can terminate if it finds that a candidate solution has zero backward error. But what if backward error is small but nonzero? Does this condition necessarily imply small forward error? We must address such questions to justify replacing forward error with backward error for evaluating the success of a numerical algorithm.

The relationship between forward and backward error can be different for each problem we wish to solve, so in the end we make the following rough classification:

- A problem is *insensitive* or *well-conditioned* when small amounts of backward error imply small amounts of forward error. In other words, a small perturbation to the statement of a well-conditioned problem yields only a small perturbation of the true solution.
- A problem is *sensitive*, *poorly conditioned*, or *stiff* when this is not the case.

Example 2.8 ($ax = b$)**.** Suppose as a toy example that we want to find the solution $x_0 \equiv b/a$ to the linear equation $ax = b$ for $a, x, b \in \mathbb{R}$. Forward error of a potential solution x is given by $|x - x_0|$ while backward error is given by $|b - ax| = |a(x - x_0)|$. So, when $|a| \gg 1$, the problem is well-conditioned since small values of backward error $a(x - x_0)$ imply even smaller values of $x - x_0$; contrastingly, when $|a| \ll 1$ the problem is ill-conditioned, since even if $a(x - x_0)$ is small, the forward error $x - x_0 \equiv 1/a \cdot a(x - x_0)$ may be large given the $1/a$ factor.

We define the *condition number* to be a measure of a problem's sensitivity:

Definition 2.4 (Condition number)**.** The *condition number* of a problem is the ratio of how much its solution changes to the amount its statement changes under small perturbations. Alternatively, it is the ratio of forward to backward error for small changes in the problem statement.

Problems with small condition numbers are well-conditioned, and thus backward error can be used safely to judge success of approximate solution techniques. Contrastingly, much smaller backward error is needed to justify the quality of a candidate solution to a problem with a large condition number.

Example 2.9 ($ax = b$, continued)**.** Continuing Example 2.8, we can compute the condition number exactly:

$$c = \frac{\text{forward error}}{\text{backward error}} = \frac{|x - x_0|}{|a(x - x_0)|} \equiv \frac{1}{|a|}.$$

Computing condition numbers usually is nearly as hard as computing forward error, and thus their exact computation is likely impossible. Even so, many times it is possible to bound or approximate condition numbers to help evaluate how much a solution can be trusted.

Example 2.10 (Root-finding)**.** Suppose that we are given a smooth function $f : \mathbb{R} \to \mathbb{R}$ and want to find roots x with $f(x) = 0$. By Taylor's theorem, $f(x + \varepsilon) \approx f(x) + \varepsilon f'(x)$ when $|\varepsilon|$ is small. Thus, an approximation of the condition number for finding the root x is given by

$$\frac{\text{forward error}}{\text{backward error}} = \frac{|(x + \varepsilon) - x|}{|f(x + \varepsilon) - f(x)|} \approx \frac{|\varepsilon|}{|\varepsilon f'(x)|} = \frac{1}{|f'(x)|}.$$

This approximation generalizes the one in Example 2.9. If we do not know x, we cannot evaluate $f'(x)$, but if we can examine the form of f and *bound* $|f'|$ near x, we have an idea of the worst-case situation.

Forward and backward error measure the *accuracy* of a solution. For the sake of scientific repeatability, we also wish to derive *stable* algorithms that produce self-consistent solutions to a class of problems. For instance, an algorithm that generates accurate solutions only one fifth of the time might not be worth implementing, even if we can use the techniques above to check whether a candidate solution is good. Other numerical methods require the client to tune several unintuitive parameters before they generate usable output and may be unstable or sensitive to changes to any of these options.

2.3 PRACTICAL ASPECTS

The theory of error analysis introduced in §2.2 will allow us to bound the quality of numerical techniques we introduce in future chapters. Before we proceed, however, it is worth noting some more practical oversights and "gotchas" that pervade implementations of numerical methods.

We purposefully introduced the largest offender early in §2.1, which we repeat in a larger font for well-deserved emphasis:

Rarely if ever should the operator == and its equivalents be used on fractional values. Instead, some *tolerance* should be used to check if numbers are equal.

Finding a suitable replacement for `==` depends on particulars of the situation. Example 2.6 shows that a method for solving $A\vec{x} = \vec{b}$ can terminate when the residual $\vec{b} - A\vec{x}$ is zero; since we do not want to check if `A*x==b` explicitly, in practice implementations will check `norm(A*x-b)<epsilon`. This example demonstrates two techniques:

- the use of *backward* error $\vec{b} - A\vec{x}$ rather than forward error to determine when to terminate, and
- checking whether backward error is less than `epsilon` to avoid the forbidden `==0` predicate.

The parameter `epsilon` depends on how accurate the desired solution must be as well as the quality of the discrete numerical system.

Based on our discussion of relative error, we can isolate another common cause of bugs in numerical software:

Beware of operations that transition between orders of magnitude, like division by small values and subtraction of similar quantities.

Catastrophic cancellation as in Example 2.4 can cause relative error to explode even if the inputs to an operation are known with near-complete certainty.

2.3.1 Computing Vector Norms

A programmer using floating-point data types and operations must be vigilant when it comes to detecting and preventing poor numerical operations. For example, consider the following code snippet for computing the norm $\|\vec{x}\|_2$ for a vector $\vec{x} \in \mathbb{R}^n$ represented as a 1D array `x[]`:

```
double normSquared = 0;
for (int i = 0; i < n; i++)
        normSquared += x[i]*x[i];
return sqrt(normSquared);
```

In theory, $\min_i |x_i| \leq {}^{\|\vec{x}\|_2}/_{\sqrt{n}} \leq \max_i |x_i|$, that is, the norm of $\vec{x}$ is on the order of the values of elements contained in $\vec{x}$. Hidden in the computation of $\|\vec{x}\|_2$, however, is the expression `x[i]*x[i]`. If there exists `i` such that `x[i]` is near `DOUBLE_MAX`, the product `x[i]*x[i]` will overflow even though $\|\vec{x}\|_2$ is still within the range of the `double`s. Such overflow is preventable by dividing $\vec{x}$ by its maximum value, computing the norm, and multiplying back:

```
double maxElement = epsilon; // don't want to divide by zero!
for (int i = 0; i < n; i++)
        maxElement = max(maxElement, fabs(x[i]));
for (int i = 0; i < n; i++) {
        double scaled = x[i] / maxElement;
        normSquared += scaled*scaled;
}
return sqrt(normSquared) * maxElement;
```

The scaling factor alleviates the overflow problem by ensuring that elements being summed are no larger than 1, at the cost of additional computation time.

This small example shows one of many circumstances in which a single *character* of code can lead to a non-obvious numerical issue, in this case the product `*`. While our intuition from continuous mathematics is sufficient to formulate many numerical methods, we must always double-check that the operations we employ are valid when transitioning from theory to finite-precision arithmetic.

```
function SIMPLE-SUM(x⃗)
  s ← 0                                  ▷ Current total
  for i ← 1, 2, ..., n : s ← s + x_i
  return s
```

(a)

```
function KAHAN-SUM(x⃗)
  s, c ← 0                               ▷ Current total and compensation
  for i ← 1, 2, ..., n
    v ← x_i + c                          ▷ Try to add x_i and compensation c to the sum
    s_next ← s + v                       ▷ Compute the summation result of this iteration
    c ← v − (s_next − s)                 ▷ Compute compensation using the Kahan error estimate
    s ← s_next                           ▷ Update sum
  return s
```

(b)

Figure 2.3 (a) A simplistic method for summing the elements of a vector $\vec{x}$; (b) the Kahan summation algorithm.

2.3.2 Larger-Scale Example: Summation

We now provide an example of a numerical issue caused by finite-precision arithmetic whose resolution involves a more subtle algorithmic trick. Suppose that we wish to sum a list of floating-point values stored in a vector $\vec{x} \in \mathbb{R}^n$, a task required by systems in accounting, machine learning, graphics, and nearly any other field. A simple strategy, iterating over the elements of $\vec{x}$ and incrementally adding each value, is detailed in Figure 2.3(a). For the vast majority of applications, this method is stable and mathematically valid, but in challenging cases it can fail.

What can go wrong? Consider the case where n is large and most of the values x_i are small and positive. Then, as i progresses, the current sum s will become large relative to x_i. Eventually, s could be so large that adding x_i would change only the lowest-order bits of s, and in the extreme case s could be large enough that adding x_i has no effect whatsoever. Put more simply, adding a long list of small numbers can result in a large sum, even if any single term of the sum appears insignificant.

To understand this effect mathematically, suppose that computing a sum $a + b$ can be off by as much as a factor of $\varepsilon > 0$. Then, the method in Figure 2.3(a) can induce error on the order of $n\varepsilon$, which grows linearly with n. If most elements x_i are on the order of ε, then the sum cannot be trusted *whatsoever*! This is a disappointing result: The error can be as large as the sum itself.

Fortunately, there are many ways to do better. For example, adding the smallest values first might make sure they are not deemed insignificant. Methods recursively adding pairs of values from $\vec{x}$ and building up a sum also are more stable, but they can be difficult to implement as efficiently as the **for** loop above. Thankfully, an algorithm by Kahan provides an easily implemented "compensated summation" method that is nearly as fast as iterating over the array [69].

The useful observation to make is that we can approximate the inaccuracy of s as it changes from iteration to iteration. To do so, consider the expression

$$((a + b) - a) - b.$$

Algebraically, this expression equals zero. Numerically, however, this may not be the case. In particular, the sum $(a+b)$ may be rounded to floating-point precision. Subtracting a and b one at a time then yields an approximation of the error of approximating $a+b$. Removing a and b from $a + b$ intuitively transitions *from* large orders of magnitude *to* smaller ones rather than vice versa and hence is less likely to induce rounding error than evaluating the sum $a+b$; this observation explains why the error estimate is not itself as prone to rounding issues as the original operation.

With this observation in mind, the Kahan technique proceeds as in Figure 2.3(b). In addition to maintaining the sum s, now we keep track of a *compensation* value c approximating the difference between s and the true sum at each iteration i. During each iteration, we attempt to add this compensation to s in addition to the current element x_i of $\vec{x}$; then we recompute c to account for the latest error.

Analyzing the Kahan algorithm requires more careful bookkeeping than analyzing the incremental technique in Figure 2.3(a). Although constructing a formal mathematical argument is outside the scope of our discussion, the final mathematical result is that error is on the order $O(\varepsilon + n\varepsilon^2)$, a considerable improvement over $O(n\varepsilon)$ when $0 \leq \varepsilon \ll 1$. Intuitively, it makes sense that the $O(n\varepsilon)$ term from Figure 2.3(a) is reduced, since the compensation attempts to represent the small values that were otherwise neglected. Formal arguments for the ε^2 bound are surprisingly involved; one detailed derivation can be found in [49].

Implementing Kahan summation is straightforward but more than doubles the operation count of the resulting program. In this way, there is an implicit trade-off between speed and accuracy that software engineers must make when deciding which technique is most appropriate. More broadly, Kahan's algorithm is one of several methods that bypass the accumulation of numerical error during the course of a computation consisting of more than one operation. Another representative example from the field of computer graphics is Bresenham's algorithm for rasterizing lines [18], which uses only integer arithmetic to draw lines even when they intersect rows and columns of pixels at non-integer locations.

2.4 EXERCISES

2.1 When might it be preferable to use a fixed-point representation of real numbers over floating-point? When might it be preferable to use a floating-point representation of real numbers over fixed-point?

DH 2.2 ("Extraterrestrial chemistry") Suppose we are programming a planetary rover to analyze the chemicals in a gas found on a neighboring planet. Our rover is equipped with a flask of volume $0.5\,\mathrm{m}^3$ and also has pressure and temperature sensors. Using the sensor readouts from a given sample, we would like our rover to determine the amount of gas our flask contains.

One of the fundamental physical equations describing a gas is the Ideal Gas Law $PV = nRT$, which states:

$$(P)\text{ressure} \cdot (V)\text{olume} = \text{amou}(n)\text{t of gas} \cdot R \cdot (T)\text{emperature},$$

where R is the ideal gas constant, approximately equal to $8.31\,\mathrm{J} \cdot \mathrm{mol}^{-1} \cdot \mathrm{K}^{-1}$. Here, P is in pascals, V is in cubic meters, n is in moles, and T is in Kelvin. We will use this equation to approximate n given the other variables.

(a) Describe any forms of rounding, discretization, modeling, and input error that can occur when solving this problem.

(b) Our rover's pressure and temperature sensors do not have perfect accuracy. Suppose the pressure and temperature sensor measurements are accurate to within $\pm\varepsilon_P$ and $\pm\varepsilon_T$, respectively. Assuming V, R, and fundamental arithmetic operations like $+$ and $\times$ induce no errors, bound the relative forward error in computing n, when $0 < \varepsilon_P \ll P$ and $0 < \varepsilon_T \ll T$.

(c) Continuing the previous part, suppose $P = 100\,\text{Pa}$, $T = 300\,\text{K}$, $\varepsilon_P = 1\,\text{Pa}$, and $\varepsilon_T = 0.5\,\text{K}$. Derive upper bounds for the worst absolute and relative errors that we could obtain from a computation of n.

(d) Experiment with perturbing the variables P and T. Based on how much your estimate of n changes between the experiments, suggest when this problem is well-conditioned or ill-conditioned.

DH 2.3 In contrast to the "absolute" condition number introduced in this chapter, we can define the "relative" condition number of a problem to be

$$\kappa_{\text{rel}} \equiv \frac{\text{relative forward error}}{\text{relative backward error}}.$$

In some cases, the relative condition number of a problem can yield better insights into its sensitivity.

Suppose we wish to evaluate a function $f : \mathbb{R} \to \mathbb{R}$ at a point $x \in \mathbb{R}$, obtaining $y \equiv f(x)$. Assuming f is smooth, compare the absolute and relative condition numbers of computing y at x. Additionally, provide examples of functions f with large and small relative condition numbers for this problem near $x = 1$.
Hint: Start with the relationship $y + \Delta y = f(x + \Delta x)$, and use Taylor's theorem to write the condition numbers in terms of x, $f(x)$, and $f'(x)$.

2.4 Suppose $f : \mathbb{R} \to \mathbb{R}$ is infinitely differentiable, and we wish to write algorithms for finding x^* minimizing $f(x)$. Our algorithm outputs x_{est}, an approximation of x^*. Assuming that in our context this problem is equivalent to finding roots of $f'(x)$, write expressions for:

(a) Forward error of the approximation

(b) Backward error of the approximation

(c) Conditioning of this minimization problem near x^*

2.5 Suppose we are given a list of floating-point values $x_1, x_2, \ldots, x_n$. The following quantity, known as their "log-sum-exp," appears in many machine learning algorithms:

$$\ell(x_1, \ldots, x_n) \equiv \ln\left[\sum_{k=1}^{n} e^{x_k}\right].$$

(a) The value $p_k \equiv e^{x_k}$ often represents a probability $p_k \in (0, 1]$. In this case, what is the range of possible x_k's?

(b) Suppose many of the x_k's are very negative ($x_k \ll 0$). Explain why evaluating the log-sum-exp formula as written above may cause numerical error in this case.

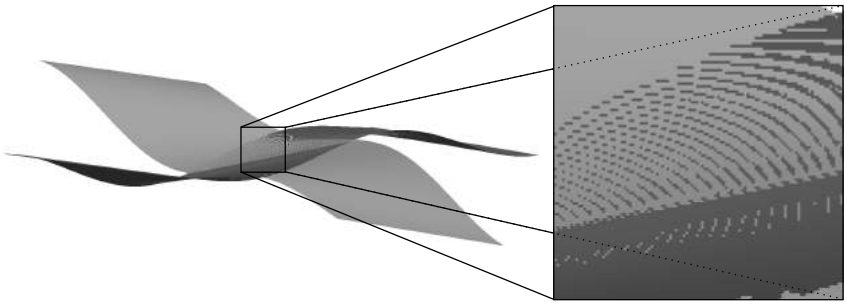

Figure 2.4 z-fighting, for Exercise 2.6; the overlap region is zoomed on the right.

(c) Show that for any $a \in \mathbb{R}$,

$$\ell(x_1, \ldots, x_n) = a + \ln \left[\sum_{k=1}^{n} e^{x_k - a} \right].$$

To avoid the issues you explained in 2.5b, suggest a value of a that may improve the stability of computing $\ell(x_1, \ldots, x_n)$.

2.6 ("z-fighting") A typical pipeline in computer graphics draws three-dimensional surfaces on the screen, one at a time. To avoid rendering a far-away surface on top of a close one, most implementations use a *z-buffer*, which maintains a double-precision depth value $z(x, y) \geq 0$ representing the depth of the closest object to the camera at each screen coordinate (x, y). A new object is rendered at (x, y) only when its z value is smaller than the one currently in the z-buffer.

A common artifact when rendering using z-buffering known as *z-fighting* is shown in Figure 2.4. Here, two surfaces overlap at some visible points. Why are there rendering artifacts in this region? Propose a strategy for avoiding this artifact; there are many possible resolutions.

2.7 (Adapted from Stanford CS 205A, 2012) Thanks to floating-point arithmetic, in most implementations of numerical algorithms we cannot expect that computations involving fractional values can be carried out with 100% precision. Instead, every time we do a numerical operation we induce the potential for error. Many models exist for studying how this error affects the quality of a numerical operation; in this problem, we will explore one common model.

Suppose we care about an operation $\diamond$ between two scalars x and y; here $\diamond$ might stand for $+$, $-$, $\times$, $\div$, and so on. As a model for the error that occurs when computing $x \diamond y$, we will say that evaluating $x \diamond y$ on the computer yields a number $(1 + \varepsilon)(x \diamond y)$ for some number ε satisfying $0 \leq |\varepsilon| < \varepsilon_{\text{max}} \ll 1$; we will assume ε can depend on $\diamond$, x, and y.

(a) Why is this a reasonable model for modeling numerical issues in floating-point arithmetic? For example, why does this make more sense than assuming that the output of evaluating $x \diamond y$ is $(x \diamond y) + \varepsilon$?

(b) (Revised by B. Jo) Suppose we are given two vectors $\vec{x}, \vec{y} \in \mathbb{R}^n$ and compute their dot product as s_n via the recurrence:

$$s_0 \equiv 0$$
$$s_k \equiv s_{k-1} + x_k y_k.$$

In practice, both the addition and multiplication steps of computing s_k from s_{k-1} induce numerical error. Use $\hat{s}_k$ to denote the actual value computed incorporating numerical error, and denote $e_k \equiv |\hat{s}_k - s_k|$. Show that

$$|e_n| \leq n\varepsilon_{\max}\bar{s}_n + O(n\varepsilon_{\max}^2\bar{s}_n),$$

where $\bar{s}_n \equiv \sum_{k=1}^{n} |x_k||y_k|$. You can assume that adding $x_1 y_1$ to zero incurs no error, so $\hat{s}_1 = (1 + \varepsilon^{\times})x_1 y_1$, where $\varepsilon^{\times}$ encodes the error induced by multiplying x_1 and y_1. You also can assume that $n\varepsilon_{\max} < 1$.

2.8 Argue using the error model from the previous problem that the relative error of computing $x - y$ for $x, y > 0$ can be unbounded; assume that there is error in representing x and y in addition to error computing the difference. This phenomenon is known as "catastrophic cancellation" and can cause serious numerical issues.

2.9 In this problem, we continue to explore the conditioning of root-finding. Suppose $f(x)$ and $p(x)$ are smooth functions of $x \in \mathbb{R}$.

(a) Thanks to inaccuracies in how we evaluate or express $f(x)$, we might accidentally compute roots of a perturbation $f(x) + \varepsilon p(x)$. Take x^* to be a root of f, so $f(x^*) = 0$. If $f'(x^*) \neq 0$, for small ε we can write a function $x(\varepsilon)$ such that $f(x(\varepsilon)) + \varepsilon p(x(\varepsilon)) = 0$, with $x(0) = x^*$. Assuming such a function exists and is differentiable, show:

$$\left.\frac{dx}{d\varepsilon}\right|_{\varepsilon=0} = -\frac{p(x^*)}{f'(x^*)}.$$

(b) Assume $f(x)$ is given by Wilkinson's polynomial [131]:

$$f(x) \equiv (x-1)\cdot(x-2)\cdot(x-3)\cdot\cdots\cdot(x-20).$$

We could have expanded $f(x)$ in the monomial basis as $f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{20}x^{20}$, for appropriate choices of $a_0, \ldots, a_{20}$. If we express the coefficient a_{19} inaccurately, we could use the model from Exercise 2.9a with $p(x) \equiv x^{19}$ to predict how much root-finding will suffer. For these choices of $f(x)$ and $p(x)$, show:

$$\left.\frac{dx}{d\varepsilon}\right|_{\varepsilon=0, x^*=j} = -\prod_{k\neq j}\frac{j}{j-k}.$$

(c) Compare $\frac{dx}{d\varepsilon}$ from the previous part for $x^* = 1$ and $x^* = 20$. Which root is more stable to this perturbation?

2.10 The roots of the quadratic function $ax^2 + bx + c$ are given by the *quadratic equation*

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

(a) Prove the alternative formula

$$x^* \in \frac{-2c}{b \pm \sqrt{b^2 - 4ac}}.$$

(b) Propose a numerically stable algorithm for solving the quadratic equation.

2.11 One technique for tracking uncertainty in a calculation is the use of *interval arithmetic*. In this system, an uncertain value for a variable x is represented as the interval $[x] \equiv [\underline{x}, \overline{x}]$ representing the range of possible values for x, from $\underline{x}$ to $\overline{x}$. Assuming infinite-precision arithmetic, give update rules for the following in terms of $\underline{x}$, $\overline{x}$, $\underline{y}$, and $\overline{y}$:

- $[x] + [y]$
- $[x] - [y]$
- $[x] \times [y]$
- $[x] \div [y]$
- $[x]^{1/2}$

Additionally, propose a conservative modification for finite-precision arithmetic.

2.12 Algorithms for dealing with geometric primitives such as line segments and triangles are notoriously difficult to implement in a numerically stable fashion. Here, we highlight a few ideas from "ε-geometry," a technique built to deal with these issues [55].

(a) Take $\vec{p}, \vec{q}, \vec{r} \in \mathbb{R}^2$. Why might it be difficult to determine whether $\vec{p}$, $\vec{q}$, and $\vec{r}$ are collinear using finite-precision arithmetic?

(b) We will say $\vec{p}$, $\vec{q}$, and $\vec{r}$ are ε-collinear if there exist $\vec{p}'$ with $\|\vec{p} - \vec{p}'\|_2 \leq \varepsilon$, $\vec{q}'$ with $\|\vec{q} - \vec{q}'\|_2 \leq \varepsilon$, and $\vec{r}'$ with $\|\vec{r} - \vec{r}'\|_2 \leq \varepsilon$ such that $\vec{p}'$, $\vec{q}'$, and $\vec{r}'$ are exactly collinear. For fixed $\vec{p}$ and $\vec{q}$, sketch the region $\{\vec{r} \in \mathbb{R}^2 : \vec{p}, \vec{q}, \vec{r} \text{ are } \varepsilon\text{-collinear}\}$. This region is known as the *ε-butterfly* of $\vec{p}$ and $\vec{q}$.

(c) An ordered triplet $(\vec{p}, \vec{q}, \vec{r}) \in \mathbb{R}^2 \times \mathbb{R}^2 \times \mathbb{R}^2$ is ε-clockwise if the three points can be perturbed by at most distance ε so that they form a triangle whose vertices are in clockwise order; we will consider collinear triplets to be both clockwise and counterclockwise. For fixed $\vec{p}$ and $\vec{q}$, sketch the region $\{\vec{r} \in \mathbb{R}^2 : (\vec{p}, \vec{q}, \vec{r}) \text{ is } \varepsilon\text{-clockwise}\}$.

(d) Show a triplet is ε-collinear if and only if it is both ε-clockwise and ε-counterclockwise.

(e) A point $\vec{x} \in \mathbb{R}^2$ is ε-inside the triangle $(\vec{p}, \vec{q}, \vec{r})$ if and only if $\vec{p}$, $\vec{q}$, $\vec{r}$, and $\vec{x}$ can be moved by at most distance ε such that the perturbed $\vec{x}'$ is exactly inside the perturbed triangle $(\vec{p}', \vec{q}', \vec{r}')$. Show that when $\vec{p}$, $\vec{q}$, and $\vec{r}$ are in (exactly) clockwise order, $\vec{x}$ is inside $(\vec{p}, \vec{q}, \vec{r})$ if and only if $(\vec{p}, \vec{q}, \vec{x})$, $(\vec{q}, \vec{r}, \vec{x})$, and $(\vec{r}, \vec{p}, \vec{x})$ are all clockwise. Is the same statement true if we relax to ε-inside and ε-clockwise?