Chapter 1

Numerics and Error Analysis

In studying numerical analysis, we move from dealing with `int`s and `long`s to `float`s and `double`s. This seemingly innocent transition comprises a huge shift in how we must think about algorithmic design and implementation. Unlike the basics of discrete algorithms, we no longer can expect our algorithms to yield exact solutions in all cases. "Big O" and operation counting do not always reign supreme; instead, even in understanding the most basic techniques we are forced to study the trade off between timing, approximation error, and so on.

1.1 Storing Numbers with Fractional Parts

Recall that computers generally store data in *binary* format. In particular, each digit of a positive integer corresponds to a different power of two. For instance, we might 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

In other words, this notation encodes the fact that 463 can be decomposed into powers of two uniquely as:

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

Issues of overflow aside, all positive integers can be written in this form using a finite number of digits. Negative numbers also can be represented this way, either by introducing a leading sign bit or by using the "two's complement" trick.

Such a decomposition inspires a simple extension to numbers that include fractions: simply include negative powers of two. For instance, decomposing 463.25 is as simple as 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}

Just as in the decimal system, however, representing fractional parts of numbers this way is not nearly as well-behaved as representing integers. For instance, writing the fraction $1/3$ in binary yields the expression:

$$\frac{1}{3} = 0.0101010101\ldots$$

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

For this reason, when designing computational systems that do math on $\mathbb{R}$ instead of $\mathbb{Z}$, we are forced to make approximations for nearly any reasonably efficient numerical representation. This can lead to many points of confusion while coding. For instance, consider the following C++ snippet:

```
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? The definition of `y` makes an approximation to $1/3$ since it cannot be written as a terminating binary string, rounding to a nearby number it can represent. Thus, `y*3.0` no longer is multiplying 3 by $1/3$. One way to fix this issue is below:

```
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 within some tolerance of one another rather than checking exact equality. This is an example of a very important point:

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.

Of course, there is a tradeoff 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.

We consider a few options for representing numbers on a computer below.

1.1.1 Fixed Point Representations

The most straightforward option for storing fractions is to add a *fixed* decimal point. That is, as in the example above we represent values by storing 0/1 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 $1/4$ is as easy as taking $k = 2$ and $\ell = 7$; in this situation, we represent these values using 9 binary digits, of which two occur after the decimal point.

The primary advantage of this representation is that nearly all arithmetic operations can be carried out using the same algorithms as with integers. For example, it is easy to see that

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

Multiplying our fixed representation by 2^k guarantees the result is integral, so this observation essentially shows that addition can be carried out using integer addition essentially by "ignoring" the decimal point. Thus, rather than using specialized hardware, the pre-existing integer arithmetic logic unit (ALU) carries out fixed-point mathematics quickly.

Fixed-point arithmetic may be fast, but it can suffer 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. In this system, it is fairly 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 decimal data type. The speed and regularity of fixed-pont arithmetic, however, can be a considerable advantage for systems that favor timing over accuracy. In fact, some lower-end graphics processing units (GPU) implement only these operations since a few decimal points of precision is sufficient for many graphical applications.

1.1.2 Floating Point Representations

One of many numerical challenges in writing scientific applications is the variety of scales that can appear. Chemists alone deal with values anywhere between 9.11×10^{-31} and 6.022×10^{23}. An operation as innocent as a change of units can cause a sudden transition between these regimes: the same observation written in kilograms per lightyear will look considerably different in megatons per second. As numerical analysts, our job is to write software that can transition between these scales gracefully without imposing on the client unnatural restrictions on their techniques.

A few notions and obsevations from the art of scientific measurement are relevant to such a discussion. First, obviously one of the following representations is more compact than the other:

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

Furthermore, in the absence of exceptional scientific equipment, the difference between 6.022×10^{23} and $6.022 \times 10^{23} + 9.11 \times 10^{-31}$ is negligible. One way to come to this conclusion is to say that 6.022×10^{23} has only three *digits of precision* and probably represents some range of possible measurements $[6.022 \times 10^{23} - \varepsilon, 6.011 \times 10^{23} + \varepsilon]$ for some $\varepsilon \sim 0.001 \times 10^{23}$.

Our first observation was able to compactify our 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 $a \times 10^b$ for some $a \sim 1$ and $b \in \mathbb{Z}$. We call this format the *floating-point* form of a number, because unlike the fixed-point setup in §1.1.1, here the decimal point "floats" to the top. We can describe floating point systems using a few parameters (CITE):

- The *base* $\beta \in \mathbb{N}$; for scientific notation explained above, the base is 10
- The *precision* $p \in \mathbb{N}$ representing the number of digits in the decimal expansion
- The range of exponents $[L, U]$ representing the possible values of b

Such an expansion looks like:

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

where each digit d_k is in the range $[0, \beta - 1]$ and $b \in [L, U]$.

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 β and β^2 is the same as that between β^2 and β^3 even though usually $\beta^3 - \beta^2 > \beta^2 - \beta$. 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. Then, numbers like $\beta + \varepsilon_m$ are not expressible in the number system because ε_m is too small!

By far the most common standard 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 $\beta = 2$ (as are most numbers on the computer), 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 $10/0$. An extra bit of precision can be gained by writing *normalizing* floating point values and assuming the most significant digit d_0 is 1 and not writing it.

The IEEE standard also includes agreed-upon options for dealing with the finite number of values that can be represented given a finite number of bits. 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. Note that there are many equally legitimate strategies for rounding; choosing a single one guarantees that scientific software will work identically on all client machines implementing the same standard.

1.1.3 More Exotic Options

Moving forward, we will assume that decimal 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 adding tolerances to account for rounding errors might be unacceptable for some applications. This situation appears in computational geometry applications, e.g. when the difference between *nearly-* and *completely*-parallel lines may be a difficult distinction to make. One resolution might be to use *arbitrary-precision arithmetic,* that is, to implement 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 $\mathbb{Q}$ are sufficient for a given application, which can be written as ratios a/b for $a, b \in \mathbb{Z}$. Basic arithmetic operations can be carried out in $\mathbb{Q}$ without any loss in precision. For instance, it is easy to see

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

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. $a/b = 5a/5b$.

Other times 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.

1.2 Understanding Error

With the exception of the arbitrary-precision systems describe in §1.1.3, nearly every computerized representation of real numbers with fractional parts is forced to employ rounding and other approximation schemes. This scheme represents one of many sources of approximations typically encountered in numerical systems:

- *Truncation* error comes from the fact that we can only represent a finite subset of all the possible set of values in $\mathbb{R}$; for example, we must truncate long or infinite sequences past the decimal point to the number of bits we are willing to store.
- *Discretization* error comes from our computerized adaptations of calculus, physics, and other aspects of continuous mathematics. For instance, we make an approximation

  $$\frac{dy}{dx} \approx \frac{y(x+\varepsilon) - y(x)}{\varepsilon}.$$

  We will learn that this approximation is a legitimate and useful one, but depending on the choice of ε it may not be completely correct.
- *Modeling* error comes from incomplete or inaccurate descriptions of the problems we wish to solve. For instance, a simulation for predicting weather in Germany may choose to neglect the collective flapping of butterfly wings in Malaysia, although the displacement of air by these butterflies may be enough to perturb the weather patterns elsewhere somewhat.
- *Empirical constant* error comes from poor representations of physical or mathematical constants. For instance, we may compute π using a Taylor sequence that we terminate early, and even scientists may not even know the speed of light to more than some number of digits.
- *Input* error can come from user-generated approximations of parameters of a given system (and from typos!). Simulation and numerical techniques can be used to answer "what if" type questions, in which exploratory choices of input setups are chosen just to get some idea of how a system behaves.

Example 1.1 (Computational physics)**.** *Suppose we are designing a system for simulating planets as they revolve around the earth. The system essentially solves Newton's equation $F = ma$ by integrating forces forward in time. Examples of error sources in this system might include:*

- *Truncation error: Using IEEE floating point to represent parameters and output of the system and truncating when computing the product ma*
- *Discretization error: Replacing the acceleration a with a divided difference*
- *Modeling error: Neglecting to simulate the moon's effects on the earth's motion within the planetary system*
- *Empirical error: Only entering the mass of Jupiter to four digits*
- *Input error: The user may wish to evaluate the cost of sending garbage into space rather than risking a Wall-E style accumulation on Earth but can only estimate the amount of garbage the government is willing to jettison in this fashion*

1.2.1 Classifying Error

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

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

Although it has the size as the range $[1 - 0.01, 1 + 0.01]$, the range $[10^5 - 0.01, 10^5 + 0.01]$ 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 differentiating between *absolute* error and *relative* error:

Definition 1.1 (Absolute error)**.** *The* absolute error *of a measurement is given by the difference between the approximate value and its underlying true value.*

Definition 1.2 (Relative error)**.** *The* relative error *of a measurement is given by the absolute error divided by the true value.*

One way to distinguish between these two species of error is the use of units versus percentages.

Example 1.2 (Absolute and relative error)**.** *Here are two equivalent statements in contrasting forms:*

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

In most applications the *true* value is unknown; after all, if this were not the case the use of an approximation in lieu of the true value may be a dubious proposition. There are two popular ways to resolve this issue. The first simply 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 that our solution is useful.

An alternative resolution has to do with *what* you can measure. For instance, suppose we wish to solve the equation $f(x) = 0$ for x given a function $f : \mathbb{R} \to \mathbb{R}$. We know that somewhere there exists a root x_0 satisfying $f(x_0) = 0$ exactly, but if we knew this root our algorithm would not be necessary in the first place. In practice, our computational system may yield some x_{est} satisfying $f(x_{est}) = \varepsilon$ for some ε with $|\varepsilon| \ll 1$. We may not be able to evaluate the difference $x_0 - x_{est}$ since x_0 is unknown. On the other hand, simply by evaluating f we can compute $f(x_{est}) - f(x_0) \equiv f(x_{est})$ since we know $f(x_0) = 0$ by definition. This value gives some notion of error for our calculation.

This example illustrates the distinction between *forward* and *backward* error. The forward error made by an approximation most likely defines our intuition for error analysis as the difference between the approximated and actual solution, but as we have discussed it is not always possible to compute. The backward error, however, has the distinguishably of being calculable but not our exact objective when solving a given problem. We can adjust our definition and interpretation of backward error as we approach different problems, but one suitable if vague definition is as follows:

Definition 1.3 (Backward error)**.** Backward error *is given by the amount a problem statement would have to change to realize a given approximation of its solution.*

This definition is somewhat obtuse, so we illustrate its use in a few examples.

Example 1.3 (Linear systems)**.** *Suppose we wish to solve the $n \times n$ linear system $A\vec{x} = \vec{b}$. Call the true solution $\vec{x}_0 \equiv A^{-1}\vec{b}$. In reality, due to truncation error and other issues, our system yields a near-solution $\vec{x}_{est}$. The forward error of this approximation obviously is measured using the difference $\vec{x}_{est} - \vec{x}_0$; in practice this value is impossible to compute since we do not know $\vec{x}_0$. In reality, $\vec{x}_{est}$ is the* exact *solution to a modified system $A\vec{x} = \vec{b}_{est}$ for $\vec{b}_{est} \equiv A\vec{x}_{est}$; thus, we might measure backward error in terms of the difference $\vec{b} - \vec{b}_{est}$. Unlike the forward error, this error is easily computable without inverting A, and it is easy to see that $\vec{x}_{est}$ is a solution to the problem exactly when backward (or forward) error is zero.*

Example 1.4 (Solving equations, CITE)**.** *Suppose we write a function for finding square roots of positive numbers that outputs $\sqrt{2} \approx 1.4$. The forward error is $|1.4 - 1.41421\cdots| \approx 0.0142$. Notice that $1.4^2 = 1.96$, so the backward error is $|1.96 - 2| = 0.04$.*

The two examples above demonstrate a larger pattern that backward error can be much easier to compute than forward error. For example, evaluating forward error in Example 1.3 required inverting a matrix A while evaluating backward error required only multiplication by A. Similarly, in Example 1.4 transitioning from forward error to backward error replaced square root computation with multiplication.

1.2.2 Conditioning, Stability, and Accuracy

In nearly any numerical problem, zero backward error implies zero forward error and vice versa. Thus, 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 nonzero but small? Does this necessarily imply small forward error? Such questions motivate the analysis of most numerical techniques whose objective is to minimize forward error but in practice only can measure backward error.

We desire to analyze changes in backward error relative to forward error so that our algorithms can say with confidence using only backward error that they have produced acceptable solutions. This relationship 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* or *poorly-conditioned* when this is not the case.

Example 1.5 ($ax = b$)**.** *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 1.4 (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.*

Example 1.6 ($ax = b$, part two)**.** *Continuing Example 1.5, we can compute the condition number exactly:*

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

In general, computing condition numbers is nearly as hard as computing forward error, and thus their exact computation is likely impossible. Even so, many times it is possible to find bounds or approximations for condition numbers to help evaluate how much a solution can be trusted.

Example 1.7 (Root-finding)**.** *Suppose that we are given a smooth function $f : \mathbb{R} \to \mathbb{R}$ and want to find values x with $f(x) = 0$. Notice that $f(x + \Delta) \approx f(x) + \Delta f'(x)$. Thus, an approximation of the condition number for finding x might be*

$$\begin{aligned}\frac{\textit{change in forward error}}{\textit{change in backward error}} &= \frac{(x + \Delta) - x}{f(x + \Delta) - f(x)} \\ &\approx \frac{\Delta}{\Delta f'(x)} \\ &= \frac{1}{f'(x)}\end{aligned}$$

Notice that this approximation aligns with the one in Example 1.6. Of course, if we do not know x we cannot evaluate $f'(x)$, but if we can look at the form of f and bound *$|f'|$ near x, we have an idea of the worst-case situation.*

Forward and backward error are measures of 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 very accurate solutions only one fifth of the time might not be worth implementing, even if we can go back using the techniques above to check whether the candidate solution is a good one.

1.3 Practical Aspects

The infinitude and density of the real numbers $\mathbb{R}$ can cause pernicious bugs while implementing numerical algorithms. While the theory of error analysis introduced in §1.2 eventually will help us put guarantees on the quality of numerical techniques introduced in future chapters, it is worth noting before we proceed a number of common mistakes and "gotchas" that pervade implementations of numerical methods.

We purposefully introduced the largest offender early in §1.1, 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.

Finding a suitable replacement for == and corresponding conditions for terminating a numerical method depends on the technique under consideration. Example 1.3 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`. Notice that 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 resolution of the numerical system at use.

A programmer making use of these 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);
```

It is easy to see that 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 on the order of `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 easily 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 removes the overflow problem by making sure that elements being summed are no larger than 1.

This small example shows one of many circumstances in which a single *character* of code can lead to a non-obvious numerical issue. While our intuition from continuous mathematics is sufficient to generate many numerical methods, we must always double-check that the operations we employ are valid from a discrete standpoint.

1.3.1 Larger-Scale Example: Summation

We now provide an example of a numerical issue caused by finite-precision arithmetic that can be resolved using a less than obvious algorithmic trick.

Suppose that we wish to sum a list of floating-point values, easily a task required by systems in accounting, machine learning, graphics, and nearly any other field. A snippet of code to accomplish this task that no-doubt appears in countless applications looks as follows:

```
double sum = 0;
for (int i = 0; i < n; i++)
        sum += x[i];
```

Before we proceed, it is worth noting that for the vast majority of applications, this is a perfectly stable and certainly mathematically valid technique.

But, what can go wrong? Consider the case where `n` is large and most of the values `x[i]` are small and positive. In this case, when `i` is large enough, the variable `sum` will be large relative to `x[i]`. Eventually, `sum` could be so large that `x[i]` affects only the lowest-order bits of `sum`, and in the extreme case `sum` could be large enough that adding `x[i]` has no effect whatsoever. While a single such mistake might not be a big deal, the accumulated effect of making this mistake repeatedly could overwhelm the amount that we can trust `sum` at all.

To understand this effect mathematically, suppose that computing a sum $a + b$ can be off by as much as $\varepsilon > 0$. Then, the method above clearly can induce error on the order of $n\varepsilon$, which grows linearly with n. In fact, 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 help account for their accumulated effect. Pairwise methods recursively adding pairs of values from `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 (CITE) provides an easily-implemented "compensated summation" method that is nearly as fast.

The useful observation here is that we actually can keep track of an approximation of the error in `sum` during a given iteration. In particular, consider the expression

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

Obviously this expression algebraically is zero. Numerically, however, this may not be the case. In particular, the sum $(a + b)$ may round the result to keep it within the realm of floating-point values. Subtracting a and b one-at-a-time then yields an approximation of the error induced by this operation; notice that the subtraction operations likely are better conditioned since moving from large numbers to small ones *adds* digits of precision due to cancellation.

Thus, the Kahan technique proceeds as follows:

```
double sum = 0;
double compensation = 0; // an approximation of the error

for (int i = 0; i < n; i++) {
        // try to add back to both x[i] and the missing part
        double nextTerm = x[i] + compensation;

        // compute the summation result of this iteration
        double nextSum = sum + nextTerm;

        // compute the compensation as the difference between the term you wished
        // to add and the actual result
        compensation = nextTerm - (nextSum - sum);

        sum = nextSum;
}
```

Instead of simply maintaining `sum`, now we keep track of `sum` as well as an approximation `compensation` of the difference between `sum` and the desired value. During each iteration, we attempt to add back

this compensation in addition to the current element of `x[]`, and then we recompute `compensation` to account for the latest error.

Analyzing the Kahan algorithm requires more careful bookkeeping than analyzing the simpler incremental technique. You will walk through one derivation of an error expression at the end of this chapter; the final mathematical result will be that error improves from $n\varepsilon$ to $\mathcal{O}(\varepsilon + n\varepsilon^2)$, a considerable improvement when $0 < \varepsilon \ll 1$.

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. Other examples include Bresenham's algorithm for rasterizing lines (CITE), which uses only integer arithmetic to draw lines even when they intersect rows and columns of pixels at non-integral locations, and the Fast Fourier Transform (CITE), which effectively uses the binary partition summation trick described above.

1.4 Problems

Problem 1.1. *Here's a problem.*