Principles of Scientific Computing
Sources of Error

David Bindel and Jonathan Goodman

last revised January 2009, printed February 25, 2009
Scientific computing usually gives inexact answers. The code \( x = \text{sqrt}(2) \) produces something that is not the mathematical \( \sqrt{2} \). Instead, \( x \) differs from \( \sqrt{2} \) by an amount that we call the error. An accurate result has a small error. The goal of a scientific computation is rarely the exact answer, but a result that is as accurate as needed. Throughout this book, we use \( A \) to denote the exact answer to some problem and \( \hat{A} \) to denote the computed approximation to \( A \). The error is \( \hat{A} - A \).

There are four primary ways in which error is introduced into a computation:

(i) \textit{Roundoff error} from inexact computer arithmetic.

(ii) \textit{Truncation error} from approximate formulas.

(iii) \textit{Termination of iterations}.

(iv) \textit{Statistical error} in Monte Carlo.

This chapter discusses the first of these in detail and the others more briefly. There are whole chapters dedicated to them later on. What is important here is to understand the likely relative sizes of the various kinds of error. This will help in the design of computational algorithms. In particular, it will help us focus our efforts on reducing the largest sources of error.

We need to understand the various sources of error to debug scientific computing software. If a result is supposed to be \( A \) and instead is \( \hat{A} \), we have to ask if the difference between \( A \) and \( \hat{A} \) is the result of a programming mistake. Some bugs are the usual kind – a mangled formula or mistake in logic. Others are peculiar to scientific computing. It may turn out that a certain way of calculating something is simply not accurate enough.

Error propagation also is important. A typical computation has several stages, with the results of one stage being the inputs to the next. Errors in the output of one stage most likely mean that the output of the next would be inexact even if the second stage computations were done exactly. It is unlikely that the second stage would produce the exact output from inexact inputs. On the contrary, it is possible to have \textit{error amplification}. If the second stage output is very sensitive to its input, small errors in the input could result in large errors in the output; that is, the error will be amplified. A method with large error amplification is \textit{unstable}.

The \textit{condition number} of a problem measures the sensitivity of the answer to small changes in its input data. The condition number is determined by the problem, not the method used to solve it. The accuracy of a solution is limited by the condition number of the problem. A problem is called \textit{ill-conditioned} if the condition number is so large that it is hard or impossible to solve it accurately enough.

A computational strategy is likely to be unstable if it has an ill-conditioned subproblem. For example, suppose we solve a system of linear differential equations using the eigenvector basis of the corresponding matrix. Finding eigenvectors of a matrix can be ill-conditioned, as we discuss in Chapter \( \text{??} \). This makes
the eigenvector approach to solving linear differential equations potentially unstable, even when the differential equations themselves are well-conditioned.

1 Relative error, absolute error, and cancellation

When we approximate \( A \) by \( \hat{A} \), the **absolute error** is \( e = \hat{A} - A \), and the **relative error** is \( \epsilon = e/A \). That is,

\[
\hat{A} = A + e \quad \text{(absolute error)}, \quad \hat{A} = A \cdot (1 + \epsilon) \quad \text{(relative error)}.
\]

For example, the absolute error in approximating \( A = \sqrt{175} \) by \( \hat{A} = 13 \) is \( e \approx 0.23 \), and the relative error is \( \epsilon \approx 0.017 < 2\% \).

If we say \( e \approx 0.23 \) and do not give \( A \), we generally do not know whether the error is large or small. Whether an absolute error much less than one is “small” often depends entirely on how units are chosen for a problem. In contrast, relative error is dimensionless, and if we know \( \hat{A} \) is within 2% of \( A \), we know the error is not too large. For this reason, relative error is often more useful than absolute error.

We often describe the accuracy of an approximation by saying how many decimal digits are correct. For example, Avogadro’s number with two digits of accuracy is \( N_0 \approx 6.0 \times 10^{23} \). We write 6.0 instead of just 6 to indicate that Avogadro’s number is closer to \( 6 \times 10^{23} \) than to \( 6.1 \times 10^{23} \) or \( 5.9 \times 10^{23} \).

With three digits the number is \( N_0 \approx 6.02 \times 10^{23} \). The difference between \( N_0 \approx 6 \times 10^{23} \) and \( N_0 \approx 6.02 \times 10^{23} \) is \( 2 \times 10^{21} \), which may seem like a lot, but the relative error is about a third of one percent.

Relative error can grow through cancellation. For example, suppose \( A = B - C \), with \( B \approx \hat{B} = 2.38 \times 10^5 \) and \( C \approx \hat{C} = 2.33 \times 10^5 \). Since the first two digits of \( B \) and \( C \) agree, then they cancel in the subtraction, leaving only one correct digit in \( A \). Doing the subtraction exactly gives \( \hat{A} = \hat{B} - \hat{C} = 5 \times 10^3 \).

The absolute error in \( A \) is just the sum of the absolute errors in \( B \) and \( C \), and probably is less than \( 10^3 \). But this gives \( \hat{A} \) a relative accuracy of less than 10%, even though the inputs \( \hat{B} \) and \( \hat{C} \) had relative accuracy a hundred times smaller. **Catastrophic cancellation** is losing many digits in one subtraction. More subtle is an accumulation of less dramatic cancellations over many steps, as illustrated in Exercise 3.

2 Computer arithmetic

For many tasks in computer science, all arithmetic can be done with integers. In scientific computing, though, we often deal with numbers that are not integers, or with numbers that are too large to fit into standard integer types. For this reason, we typically use **floating point** numbers, which are the computer version of numbers in scientific notation.
2.1 Bits and ints

The basic unit of computer storage is a bit (binary digit), which may be 0 or 1. Bits are organized into 32-bit or 64-bit words. There $2^{32} \approx$ four billion possible 32-bit words; a modern machine running at 2-3 GHz could enumerate them in a second or two. In contrast, there are $2^{64} \approx 1.8 \times 10^{19}$ possible 64-bit words; to enumerate them at the same rate would take more than a century.

C++ has several basic integer types: short, int, and long int. The language standard does not specify the sizes of these types, but most modern systems have a 16-bit short, and a 32-bit int. The size of a long is 32 bits on some systems and 64 bits on others. For portability, the C++ header file cstdint (or the C header stdint.h) defines types int16_t, int32_t, and int64_t that are exactly 8, 16, 32, and 64 bits.

An ordinary $b$-bit integer can take values in the range $-2^{b-1}$ to $2^{b-1} - 1$; an unsigned $b$-bit integer (such as an unsigned int) takes values in the range 0 to $2^b - 1$. Thus a 32-bit integer can be between $-2^{31}$ and $2^{31} - 1$, or between about -2 billion and +2 billion. Integer addition, subtraction, and multiplication are done exactly when the results are within the representable range, and integer division is rounded toward zero to obtain an integer result. For example, $(-7)/2$ produces -3.

When integer results are out of range (an overflow), the answer is not defined by the standard. On most platforms, the result will be wrap around. For example, if we set a 32-bit int to $2^{31} - 1$ and increment it, the result will usually be $-2^{31}$. Therefore, the loop

```c
for (int i = 0; i < 2e9; ++i);
```

takes seconds, while the loop

```c
for (int i = 0; i < 3e9; ++i);
```

never terminates, because the number $3e9$ (three billion) is larger than any number that can be represented as an int.

2.2 Floating point basics

Floating point numbers are computer data-types that represent approximations to real numbers rather than integers. The IEEE floating point standard is a set of conventions for computer representation and processing of floating point numbers. Modern computers follow these standards for the most part. The standard has three main goals:

1. To make floating point arithmetic as accurate as possible.
2. To produce sensible outcomes in exceptional situations.
3. To standardize floating point operations across computers.
Floating point numbers are like numbers in ordinary scientific notation. A number in scientific notation has three parts: a sign, a mantissa in the interval \([1,10)\), and an exponent. For example, if we ask MATLAB to display the number \(-2752 = -2.572 \times 10^3\) in scientific notation (using \texttt{format short e}), we see

\[-2.7520e+03\]

For this number, the sign is negative, the mantissa is 2.7520, and the exponent is 3.

Similarly, a normal binary floating point number consists of a sign \(s\), a mantissa \(1 \leq m < 2\), and an exponent \(e\). If \(x\) is a floating point number with these three fields, then the value of \(x\) is the real number

\[
\text{val}(x) = (-1)^s \times 2^e \times m. \tag{2}
\]

For example, we write the number \(-2752 = -2.752 \times 10^3\) as

\[
-2752 = (-1)^1 \times (2^{11} + 2^9 + 2^7 + 2^6)
= (-1)^1 \times 2^{11} \times (1 + 2^{-2} + 2^{-4} + 2^{-5})
= (-1)^1 \times 2^{11} \times (1 + (.01)_2 + (.0001)_2)
-2752 = (-1)^3 \times 2^{11} \times (1.01011)_2.
\]

The bits in a floating point word are divided into three groups. One bit represents the sign: \(s = 1\) for negative and \(s = 0\) for positive, according to (2). There are \(p - 1\) bits for the mantissa and the rest for the exponent. For example (see Figure 1), a 32-bit single precision floating point word has \(p = 24\), so there are 23 mantissa bits, one sign bit, and 8 bits for the exponent.

Floating point formats allow a limited range of exponents (\(emin \leq e \leq emax\)). Note that in single precision, the number of possible exponents \([-126, -125, \ldots, 126, 127]\), is 254, which is two less than the number of 8 bit combinations \((2^8 = 256)\). The remaining two exponent bit strings (all zeros and all ones) have different interpretations described in Section 2.4. The other floating point formats, double precision and extended precision, also reserve the all zero and all one exponent bit patterns.

The mantissa takes the form

\[m = (1.b_1b_2b_3 \ldots b_{p-1})_2,\]

where \(p\) is the total number of bits (binary digits)\(^1\) used for the mantissa. In Figure 1, we list the exponent range for IEEE single precision (\texttt{float} in C/C++), IEEE double precision (\texttt{double} in C/C++), and the extended precision on the Intel processors (\texttt{long double} in C/C++).

Not every number can be exactly represented in binary floating point. For example, just as \(1/3 = .333\) cannot be written exactly as a finite decimal fraction, \(1/3 = (.010101)_2\) also cannot be written exactly as a finite binary fraction.

\(^1\)Because the first digit of a normal floating point number is always one, it is not stored explicitly.
If $x$ is a real number, we write $\hat{x} = \text{round}(x)$ for the floating point number (of a given format) that is closest\(^2\) to $x$. Finding $\hat{x}$ is called rounding. The difference $\text{round}(x) - x = \hat{x} - x$ is rounding error. If $x$ is in the range of normal floating point numbers ($2^{\text{emin}} \leq x < 2^{\text{emax}+1}$), then the closest floating point number to $x$ has a relative error not more than $|e| \leq \epsilon_{\text{mach}}$, where the machine epsilon $\epsilon_{\text{mach}} = 2^{-p}$ is half the distance between 1 and the next floating point number.

The IEEE standard for arithmetic operations (addition, subtraction, multiplication, division, square root) is: the exact answer, correctly rounded. For example, the statement $z = x \ast y$ gives $z$ the value $\text{round}(\text{val}(x) \ast \text{val}(y))$. That is: interpret the bit strings $x$ and $y$ using the floating point standard (2), perform the operation (multiplication in this case) exactly, then round the result to the nearest floating point number. For example, the result of computing $1/($float$)3$ in single precision is

$$(1.01010101010101010101011)_2 \times 2^{-2}.$$  

Some properties of floating point arithmetic follow from the above rule. For example, addition and multiplication are commutative: $x \ast y = y \ast x$. Division by powers of 2 is done exactly if the result is a normalized number. Division by 3 is rarely exact. Integers, not too large, are represented exactly. Integer arithmetic (excluding division and square roots) is done exactly. This is illustrated in Exercise 8.

Double precision floating point has smaller rounding errors because it has more mantissa bits. It has roughly 16 digit accuracy ($2^{-53} \sim 10^{-16}$, as opposed to roughly 7 digit accuracy for single precision. It also has a larger range of values. The largest double precision floating point number is $2^{1023} \sim 10^{307}$, as opposed to $2^{126} \sim 10^{38}$ for single. The hardware in many processor chips does arithmetic and stores intermediate results in extended precision, see below.

Rounding error occurs in most floating point operations. When using an unstable algorithm or solving a very sensitive problem, even calculations that would give the exact answer in exact arithmetic may give very wrong answers in floating point arithmetic. Being exactly right in exact arithmetic does not imply being approximately right in floating point arithmetic.

### 2.3 Modeling floating point error

Rounding error analysis models the generation and propagation of rounding

\(^2\) If $x$ is equally close to two floating point numbers, the answer is the number whose last bit is zero.
errors over the course of a calculation. For example, suppose \( x, y, \) and \( z \) are floating point numbers, and that we compute \( \text{fl}(x + y + z) \), where \( \text{fl}(\cdot) \) denotes the result of a floating point computation. Under IEEE arithmetic,

\[
\text{fl}(x + y) = \text{round}(x + y) = (x + y)(1 + \epsilon_1),
\]

where \(|\epsilon_1| < \epsilon_{\text{mach}}\). A sum of more than two numbers must be performed pairwise, and usually from left to right. For example:

\[
\begin{align*}
\text{fl}(x + y + z) & = \text{round}\{\text{round}(x + y) + z\} \\
& = ((x + y)(1 + \epsilon_1) + z)(1 + \epsilon_2) \\
& = (x + y + z)(1 + \epsilon_1 + (x + y + z)\epsilon_2 + (x + y)\epsilon_1\epsilon_2)
\end{align*}
\]

Here and below we use \( \epsilon_1, \epsilon_2, \) etc. to represent individual rounding errors.

It is often replace exact formulas by simpler approximations. For example, we neglect the product \( \epsilon_1\epsilon_2 \) because it is smaller than either \( \epsilon_1 \) or \( \epsilon_2 \) (by a factor of \( \epsilon_{\text{mach}} \)). This leads to the useful approximation

\[
\text{fl}(x + y + z) \approx (x + y + z) + (x + y)\epsilon_1 + (x + y + z)\epsilon_2,
\]

We also neglect higher terms in Taylor expansions. In this spirit, we have:

\[
\begin{align*}
(1 + \epsilon_1)(1 + \epsilon_2) & \approx 1 + \epsilon_1 + \epsilon_2 \\
\sqrt{1 + \epsilon} & \approx 1 + \epsilon/2.
\end{align*}
\]

As an example, we look at computing the smaller root of \( x^2 - 2x + \delta = 0 \) using the quadratic formula

\[
x = 1 - \sqrt{1 - \delta}.
\]

The two terms on the right are approximately equal when \( \delta \) is small. This can lead to catastrophic cancellation. We will assume that \( \delta \) is so small that (4) applies to (5), and therefore \( x \approx \delta/2 \).

We start with the rounding errors from the \( 1 - \delta \) subtraction and square root. We simplify with (3) and (4):

\[
\begin{align*}
\text{fl}(\sqrt{1 - \delta}) & = \left(\sqrt{(1 - \delta)(1 + \epsilon_1)}\right)(1 + \epsilon_2) \\
& \approx (\sqrt{1 - \delta})(1 + \epsilon_1/2 + \epsilon_2) = (\sqrt{1 - \delta})(1 + \epsilon_d),
\end{align*}
\]

where \( |\epsilon_d| = |\epsilon_1/2 + \epsilon_2| \leq 1.5\epsilon_{\text{mach}} \). This means that relative error at this point is of the order of machine precision but may be as much as 50% larger.

Now, we account for the error in the second subtraction\(^3\), using \( \sqrt{1 - \delta} \approx 1 \) and \( x \approx \delta/2 \) to simplify the error terms:

\[
\begin{align*}
\text{fl}(1 - \sqrt{1 - \delta}) & \approx \left(1 - (\sqrt{1 - \delta})(1 + \epsilon_d)\right)(1 + \epsilon_3) \\
& = x\left(1 - \frac{\sqrt{1 - \delta}}{x}\epsilon_d + \epsilon_3\right) \approx x\left(1 + \frac{2\epsilon_d}{\delta} + \epsilon_3\right).
\end{align*}
\]

\(^3\) For \( \delta \leq 0.75 \), this subtraction actually contributes no rounding error, since subtraction of floating point values within a factor of two of each other is exact. Nonetheless, we will continue to use our model of small relative errors in this step for the current example.
Therefore, for small $\delta$ we have

$$\hat{x} - x \approx x \frac{\epsilon}{x},$$

which says that the relative error from using the formula (5) is amplified from $\epsilon_{\text{mach}}$ by a factor on the order of $1/x$. The catastrophic cancellation in the final subtraction leads to a large relative error. In single precision with $x = 10^{-5}$, for example, we would have relative error on the order of $8\epsilon_{\text{mach}}/x \approx 0.2$, when $x$ would only expect one or two correct digits in this computation.

In this case and many others, we can avoid catastrophic cancellation by rewriting the basic formula. In this case, we could replace (5) by the mathematically equivalent $x = \delta/(1 + \sqrt{1 - \delta})$, which is far more accurate in floating point.

### 2.4 Exceptions

The smallest normal floating point number in a given format is $2^{e_{\text{min}}}$. When a floating point operation yields a nonzero number less than $2^{e_{\text{min}}}$, we say there has been an underflow. The standard formats can represent some values less than the $2^{e_{\text{min}}}$ as denormalized numbers. These numbers have the form

$$(-1)^2 \times 2^{e_{\text{min}}} \times (0.d_1 d_2 \ldots d_{p-1})_2.$$  

Floating point operations that produce results less than about $2^{e_{\text{min}}}$ in magnitude are rounded to the nearest denormalized number. This is called gradual underflow. When gradual underflow occurs, the relative error in the result may be greater than $\epsilon_{\text{mach}}$, but it is much better than if the result were rounded to 0 or $2^{e_{\text{min}}}$.

With denormalized numbers, every floating point number except the largest in magnitude has the property that the distances to the two closest floating point numbers differ by no more than a factor of two. Without denormalized numbers, the smallest number to the right of $2^{e_{\text{min}}}$ would be $2^{p-1}$ times closer than the largest number to the left of $2^{e_{\text{min}}}$; in single precision, that is a difference of a factor of about eight billion! Gradual underflow also has the consequence that two floating point numbers are equal, $x = y$, if and only if subtracting one from the other gives exactly zero.

In addition to the normal floating point numbers and the denormalized numbers, the IEEE standard has encodings for $\pm \infty$ and Not a Number (NaN). When we print these values to the screen, we see “\texttt{inf}” and “\texttt{NaN},” respectively. A floating point operation results in an \texttt{inf} if the exact result is larger than the largest normal floating point number (overflow), or in cases like $1/0$ or $\cot(0)$ where the exact result is infinite. Invalid operations such as $\sqrt{(-1.)}$ and

---

4 The actual text varies from system to system. The Microsoft Visual Studio compilers print \texttt{Ind} rather than \texttt{NaN}, for example.

5 IEEE arithmetic distinguishes between positive and negative zero, so actually $1/+0.0 = \texttt{inf}$ and $1/-0.0 = -\texttt{inf}$. 

---
log(-4.) produce NaN. Any operation involving a NaN produces another NaN. Operations with inf are common sense: inf + finite = inf, inf/inf = NaN, finite/inf = 0, inf + inf = inf, inf - inf = NaN.

A floating point operation generates an exception if the exact result is not a normalized floating point number. The five types of exceptions are an inexact result (i.e. when there is a nonzero rounding error), an underflow, an overflow, an exact infinity, or an invalid operations. When one of these exceptions occurs, a flag is raised (i.e. a bit in memory is set to one). There is one flag for each type of exception. The C99 standard defines functions to raise, lower, and check the floating point exception flags.

3 Truncation error

Truncation error is the error in analytical approximations such as

\[ f'(x) \approx \frac{f(x+h) - f(x)}{h} \]  \hspace{1cm} (6)

This is not an exact formula for \( f' \), but it can be a useful approximation. We often think of truncation error as arising from truncating a Taylor series. In this case, the Taylor series formula,

\[ f(x+h) = f(x) + hf'(x) + \frac{1}{2}h^2f''(x) + \cdots \]

is truncated by neglecting all the terms after the first two on the right. This leaves the approximation

\[ f(x+h) \approx f(x) + hf'(x) \]

which can be rearranged to give (6). Truncation usually is the main source of error in numerical integration or solution of differential equations. The analysis of truncation error using Taylor series will occupy the next two chapters.

As an example, we take \( f(x) = xe^x, \ x = 1 \), and several \( h \) values. The truncation error is

\[ e_{tr} = \frac{f(x+h) - f(x)}{h} - f'(x) \]

In Chapter ?? we will see that (in exact arithmetic) \( e_{tr} \) roughly is proportional to \( h \) for small \( h \). This is apparent in Figure 3. As \( h \) is reduced from \( 10^{-2} \) to \( 10^{-5} \) (a factor of \( 10^{-3} \)), the error decreases from \( 4.10 \times 10^{-2} \) to \( 4.08 \times 10^{-5} \), approximately the same factor of \( 10^{-3} \).

The numbers in Figure 3 were computed in double precision floating point arithmetic. The total error, \( e_{tot} \), is a combination of truncation and roundoff error. Roundoff error is significant for the smallest \( h \) values: for \( h = 10^{-8} \) the error is no longer proportional to \( h \); by \( h = 10^{-10} \) the error has increased. Such small \( h \) values are rare in a practical calculation.
<table>
<thead>
<tr>
<th>$h$</th>
<th>.3</th>
<th>.01</th>
<th>$10^{-5}$</th>
<th>$10^{-8}$</th>
<th>$10^{-10}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f'$</td>
<td>6.84</td>
<td>5.48</td>
<td>5.4366</td>
<td>5.43654</td>
<td>5.436562</td>
</tr>
<tr>
<td>$e_{\text{tot}}$</td>
<td>1.40</td>
<td>$4.10 \times 10^{-2}$</td>
<td>$4.08 \times 10^{-5}$</td>
<td>$-5.76 \times 10^{-8}$</td>
<td>$-1.35 \times 10^{-16}$</td>
</tr>
</tbody>
</table>

Figure 2: Estimates of $f'(x)$ using (6). The error is $e_{\text{tot}}$, which results from truncation and roundoff error. Roundoff error is apparent only in the last two columns.

<table>
<thead>
<tr>
<th>$n$</th>
<th>1</th>
<th>3</th>
<th>6</th>
<th>10</th>
<th>20</th>
<th>67</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_n$</td>
<td>1.46</td>
<td>1.80</td>
<td>1.751</td>
<td>1.74555</td>
<td>1.745528</td>
<td></td>
</tr>
<tr>
<td>$e_n$</td>
<td>-.745</td>
<td>-.277</td>
<td>$5.5 \times 10^{-2}$</td>
<td>$5.9 \times 10^{-3}$</td>
<td>$2.3 \times 10^{-6}$</td>
<td>$3.1 \times 10^{-17}$</td>
</tr>
</tbody>
</table>

Figure 3: Iterates of $x_{n+1} = \ln(y) - \ln(x_n)$ illustrating convergence to a limit that satisfies the equation $xe^x = y$. The error is $e_n = x_n - x$. Here, $y = 10$.

## 4 Iterative methods

A *direct method* computes the exact answer to a problem in finite time, assuming exact arithmetic. The only error in direct methods is roundoff error. However, many problems cannot be solved by direct methods. An example is finding $A$ that satisfies an equation that has no explicit solution formula. An *iterative method* constructs a sequence of approximate solutions, $A_n$, for $n = 1, 2, \ldots$. Hopefully, the approximations converge to the right answer: $A_n \to A$ as $n \to \infty$.

In practice, we must stop the iteration for some large but finite $n$ and accept $A_n$ as the approximate answer.

For example, suppose we have a $y > 0$ and we want to find $x$ such that $xe^x = y$. There is no formula for $x$, but we can write a program to carry out the iteration: $x_1 = 1$, $x_{n+1} = \ln(y) - \ln(x_n)$. The numbers $x_n$ are *iterates*. The limit $x = \lim_{n \to \infty} x_n$ (if it exists), is a *fixed point* of the iteration, i.e. $x = \ln(y) - \ln(x)$, which implies that $xe^x = y$. Figure 3 demonstrates the convergence of the iterates in this case with $y = 10$. The initial guess is $x_1 = 1$. After 20 iterations, we have $x_{20} \approx 1.75$. The relative error is $e_{20} \approx 2.3 \times 10^{-5}$, which might be small enough, depending on the application.

After 67 iterations, the relative error is $(x_{67} - x)/x \approx 3.1 \times 10^{-17}/1.75 \approx 1.8 \times 10^{-17}$, which means that $x_{67}$ is the correctly rounded result in double precision with $\epsilon_{\text{mach}} \approx 1.1 \times 10^{-16}$. This shows that supposedly approximate iterative methods can be as accurate as direct methods in floating point arithmetic. For example, if the equation were $e^x = y$, the direct method would be $x = \log(y)$. It is unlikely that the relative error resulting from the direct formula would be significantly smaller than the error we could get from an iterative method.

Chapter ?? explains many features of Figure 3. The *convergence rate* is the rate of decrease of $e_n$. We will be able to calculate it and to find methods that converge faster. In some cases an iterative method may fail to converge even though the solution is a fixed point of the iteration.
5 Statistical error in Monte Carlo

Monte Carlo means using random numbers as a computational tool. For example, suppose \( A = E[X] \), where \( X \) is a random variable with some known distribution. Sampling \( X \) means using the computer random number generator to create independent random variables \( X_1, X_2, \ldots \), each with the distribution of \( X \). The simple Monte Carlo method would be to generate \( n \) such samples and calculate the sample mean:

\[
A \approx \hat{A} = \frac{1}{n} \sum_{k=1}^{n} X_k.
\]

The difference between \( \hat{A} \) and \( A \) is statistical error. A theorem in probability, the law of large numbers, implies that \( \hat{A} \to A \) as \( n \to \infty \). Monte Carlo statistical errors typically are larger than roundoff or truncation errors. This makes Monte Carlo a method of last resort, to be used only when other methods are not practical.

Figure 4 illustrates the behavior of this Monte Carlo method for the random variable \( X = \frac{3}{2} U^2 \) with \( U \) uniformly distributed in the interval \([0, 1]\). The exact answer is \( A = E[X] = \frac{3}{2} E[U^2] = .5 \). The value \( n = 10^6 \) is repeated to illustrate the fact that statistical error is random (see Chapter ?? for a clarification of this). The errors even with a million samples are much larger than those in the right columns of Figures 3 and 3.

6 Error propagation and amplification

Errors can grow as they propagate through a computation. For example, consider the divided difference (6):

\[
f_1 = \ldots ; \quad \text{// approx of } f(x) \\
f_2 = \ldots ; \quad \text{// approx of } f(x+h) \\
fPrimeHat = (f_2 - f_1) / h ; \quad \text{// approx of derivative}
\]

There are three contributions to the final error in \( f' \):

\[
\hat{f'} = f'(x)(1 + \epsilon_{pr})(1 + \epsilon_{tr})(1 + \epsilon_r) \approx f'(x)(1 + \epsilon_{pr} + \epsilon_{tr} + \epsilon_r).
\]

\( ^6 \) \( E[X] \) is the expected value of \( X \). Chapter ?? has some review of probability.
It is unlikely that $f_1 = \hat{f}(x) \approx f(x)$ is exact. Many factors may contribute to the errors $e_1 = f_1 - f(x)$ and $e_2 = f_2 - f(x + h)$, including inaccurate $x$ values and roundoff in the code to evaluate $f$. The propagated error comes from using inexact values of $f(x + h)$ and $f(x)$:

$$\frac{f_2 - f_1}{h} = \frac{f(x + h) - f(x)}{h} \left(1 + \frac{e_2 - e_1}{f_2 - f_1}\right) = \frac{f(x + h) - f(x)}{h}(1 + \epsilon_{pr}). \quad (8)$$

The truncation error in the difference quotient approximation is

$$\frac{f(x + h) - f(x)}{h} = f'(x)(1 + \epsilon_{tr}). \quad (9)$$

Finally, there is roundoff error in evaluating $(f_2 - f_1) / h$:

$$\hat{f} = \frac{f_2 - f_1}{h}(1 + \epsilon_r). \quad (10)$$

If we multiply the errors from (8)–(10) and simplify, we get (7).

Most of the error in this calculation comes from truncation error and propagated error. The subtraction and the division in the evaluation of the divided difference each have relative error of at most $\epsilon_{mach}$; thus, the roundoff error $\epsilon_r$ is at most about $2\epsilon_{mach}$, which is relatively small$^7$. We noted in Section 3 that truncation error is roughly proportional to $h$. The propagated error $\epsilon_{pr}$ in the outputs is roughly proportional to the absolute input errors $e_1$ and $e_2$ amplified by a factor of $h^{-1}$:

$$\epsilon_{pr} = \frac{e_2 - e_1}{f_2 - f_1} \approx \frac{e_2 - e_1}{f'(x)h}.$$  

Even if $\epsilon_1 = e_1 / f(x)$ and $\epsilon_2 = e_2 / f(x + h)$ are small, $\epsilon_{pr}$ may be quite large. This increase in relative error by a large factor in one step is another example of catastrophic cancellation, which we described in Section 1. If the numbers $f(x)$ and $f(x + h)$ are nearly equal, the difference can have much less relative accuracy than the numbers themselves. More subtle is gradual error growth over many steps. Exercise 15 has an example in which the error roughly doubles at each stage. Starting from double precision roundoff level, the error after 30 steps is negligible, but the error after 60 steps is larger than the answer.

An algorithm is unstable if its error mainly comes from amplification. This numerical instability can be hard to discover by standard debugging techniques that look for the first place something goes wrong, particularly if there is gradual error growth.

In scientific computing, we use stability theory, or the study of propagation of small changes by a process, to search for error growth in computations. In a typical stability analysis, we focus on propagated error only, ignoring the original sources of error. For example, Exercise 8 involves the backward recurrence $f_{k-1} = f_{k+1} - f_k$. In our stability analysis, we assume that the subtraction is performed exactly and that the error in $f_{k-1}$ is entirely due to errors in $f_k$

$^7$ If $h$ is a power of two and $f_2$ and $f_1$ are within a factor of two of each other, then $\epsilon_r = 0.$
and \( f_{k+1} \). That is, if \( \hat{f}_k = f_k + e_k \) is the computer approximation, then the \( e_k \) satisfy the error propagation equation \( e_{k-1} = e_{k+1} - e_k \). We then would use the theory of recurrence relations to see whether the \( e_k \) can grow relative to the \( f_k \) as \( k \) decreases. If this error growth is possible, it will usually happen.

7 Condition number and ill-conditioned problems

The condition number of a problem measures the sensitivity of the answer to small changes in the data. If \( \kappa \) is the condition number, then we expect relative error at least \( \kappa \epsilon_{\text{mach}} \), regardless of the algorithm. A problem with large condition number is ill-conditioned. For example, if \( \kappa > 10^7 \), then there probably is no algorithm that gives anything like the right answer in single precision arithmetic. Condition numbers as large as \( 10^7 \) or \( 10^{16} \) can and do occur in practice.

The definition of \( \kappa \) is simplest when the answer is a single number that depends on a single scalar variable, \( x \): \( A = A(x) \). A change in \( x \) causes a change in \( A \): \( \Delta A = A(x + \Delta x) - A(x) \). The condition number measures the relative change in \( A \) caused by a small relative change of \( x \):

\[
\frac{\Delta A}{A} \approx \kappa \frac{\Delta x}{x} .
\] (11)

Any algorithm that computes \( A(x) \) must round \( x \) to the nearest floating point number, \( \hat{x} \). This creates a relative error (assuming \( x \) is within the range of normalized floating point numbers) of \(|\Delta x/x| = |(\hat{x} - x)/x| \sim \epsilon_{\text{mach}}\). If the rest of the computation were done exactly, the computed answer would be \( \hat{A}(x) = A(\hat{x}) \) and the relative error would be (using (11))

\[
\left| \frac{\hat{A}(x) - A(x)}{A(x)} \right| = \left| \frac{A(\hat{x}) - A(x)}{A(x)} \right| \approx \kappa \frac{\Delta x}{x} \sim \kappa \epsilon_{\text{mach}} .
\] (12)

If \( A \) is a differentiable function of \( x \) with derivative \( A'(x) \), then, for small \( \Delta x \), \( \Delta A \approx A'(x) \Delta x \). With a little algebra, this gives (11) with

\[
\kappa = \left| A'(x) \cdot \frac{x}{A(x)} \right| .
\] (13)

An algorithm is unstable if relative errors in the output are much larger than relative errors in the input. This analysis argues that any computation for an ill-conditioned problem must be unstable. Even if \( A(x) \) is evaluated exactly, relative errors in the input of size \( \epsilon \) are amplified by a factor of \( \kappa \). The formulas (12) and (13) represent a lower bound for the accuracy of any algorithm. An ill-conditioned problem is not going to be solved accurately, unless it can be reformulated to improve the conditioning.

A backward stable algorithm is as stable as the condition number allows. This sometimes is stated as \( \hat{A}(x) = A(\tilde{x}) \) for some \( \tilde{x} \) that is within on the
order of $\epsilon_{\text{mach}}$ of $x$, $|\tilde{x} - x|/|x| \lesssim C\epsilon_{\text{mach}}$ where $C$ is a modest constant. That is, the computer produces a result, $\hat{A}$, that is the exact correct answer to a problem that differs from the original one only by roundoff error in the input. Many algorithms in computational linear algebra in Chapter ?? are backward stable in roughly this sense. This means that they are unstable only when the underlying problem is ill-conditioned.

The condition number (13) is dimensionless because it measures relative sensitivity. The extra factor $x/A(x)$ removes the units of $x$ and $A$. Absolute sensitivity is just $A'(x)$. Note that both sides of our starting point (11) are dimensionless with dimensionless $\kappa$.

As an example consider the problem of evaluating $A(x) = R \sin(x)$. The condition number formula (13) gives

$$\kappa(x) = \left| \frac{\cos(x)}{\sin(x)} \right|.$$ 

Note that the problem remains well-conditioned ($\kappa$ is not large) as $x \to 0$, even though $A(x)$ is small when $x$ is small. For extremely small $x$, the calculation could suffer from underflow. But the condition number blows up as $x \to \pi$, because small relative changes in $x$ lead to much larger relative changes in $A$. This illustrates quirk of the condition number definition: typical values of $A$ have the order of magnitude $R$ and we can evaluate $A$ with error much smaller than this, but certain individual values of $A$ may not be computed to high relative precision. In most applications that would not be a problem.

There is no perfect definition of condition number for problems with more than one input or output. Suppose at first that the single output $A(x)$ depends on $n$ inputs $x = (x_1, \ldots, x_n)$. Of course $A$ may have different sensitivities to different components of $x$. For example, $\Delta x_1/x_1 = 1\%$ may change $A$ much more than $\Delta x_2/x_2 = 1\%$. If we view (11) as saying that $|\Delta A/A| \approx \kappa \epsilon$ for $|\Delta x/x| = \epsilon$, a worst case multicomponent generalization could be

$$\kappa = \frac{1}{\epsilon} \max \left| \frac{\Delta A}{A} \right| , \quad \text{where} \quad \left| \frac{\Delta x_k}{x_k} \right| \leq \epsilon \quad \text{for all} \quad k.$$ 

We seek the worst case $^8 \Delta x$. For small $\epsilon$, we write

$$\Delta A \approx \sum_{k=1}^{n} \frac{\partial A}{\partial x_k} \Delta x_k ,$$ 

then maximize subject to the constraint $|\Delta x_k| \leq \epsilon |x_k|$ for all $k$. The maximum occurs at $\Delta x_k = \pm \epsilon x_k$, which (with some algebra) leads to one possible generalization of (13):

$$\kappa = \sum_{k=1}^{n} \left| \frac{\partial A}{\partial x_k} \cdot \frac{x_k}{A} \right| .$$ 

\footnote{As with rounding, typical errors tend to be on the order of the worst case error.}
This formula is useful if the inputs are known to similar relative accuracy, which could happen even when the $x_k$ have different orders of magnitude or different units. Condition numbers for multivariate problems are discussed using matrix norms in Section ?? . The analogue of (13) is (??).

8 Software

Each chapter of this book has a Software section. Taken together they form a mini-course in software for scientific computing. The material ranges from simple tips to longer discussions of bigger issues. The programming exercises illustrate the chapter’s software principles as well as the mathematical material from earlier sections.

Scientific computing projects fail because of bad software as often as they fail because of bad algorithms. The principles of scientific software are less precise than the mathematics of scientific computing, but are just as important. Like other programmers, scientific programmers should follow general software engineering principles of modular design, documentation, and testing. Unlike other programmers, scientific programmers must also deal with the approximate nature of their computations by testing, analyzing, and documenting the error in their methods, and by composing modules in a way that does not needlessly amplify errors. Projects are handsomely rewarded for extra efforts and care taken to do the software “right.”

8.1 General software principles

This is not the place for a long discussion of general software principles, but it may be worthwhile to review a few that will be in force even for the small programs called for in the exercises below. In our classes, students are required to submit the program source as well the output, and points are deducted for failing to follow basic software protocols listed here.

Most software projects are collaborative. Any person working on the project will write code that will be read and modified by others. He or she will read code that will be read and modified by others. In single person projects, that “other” person may be you, a few months later, so you will end up helping yourself by coding in a way friendly to others.

• Make code easy for others to read. Choose helpful variable names, not too short or too long. Indent the bodies of loops and conditionals. Align similar code to make it easier to understand visually. Insert blank lines to separate “paragraphs” of code. Use descriptive comments liberally; these are part of the documentation.

• Code documentation includes comments and well chosen variable names. Larger codes need separate documents describing how to use them and how they work.
void do_something(double tStart, double tFinal, int n)
{
    double dt = ( tFinal - tStart ) / n; // Size of each step
    for (double t = tStart; t < tFinal; t += dt) {
        // Body of loop
    }
}

Figure 5: A code fragment illustrating a pitfall of using a floating point variable to regulate a for loop.

- Design large computing tasks into modules that perform sub-tasks.
- A code design includes a plan for building it, which includes a plan for testing. Plan to create separate routines that test the main components of your project.
- Know and use the tools in your programming environment. This includes code specific editors, window based debuggers, formal or informal version control, and make or other building and scripting tools.

8.2 Coding for floating point

Programmers who forget the inexactness of floating point may write code with subtle bugs. One common mistake is illustrated in Figure 5. For tFinal > tStart, this code would give the desired n iterations in exact arithmetic. Because of rounding error, though, the variable t might be above or below tFinal, and so we do not know whether this code while execute the loop body n times or n + 1 times. The routine in Figure 5 has other issues, too. For example, if tFinal <= tStart, then the loop body will never execute. The loop will also never execute if either tStart or tFinal is a NaN, since any comparison involving a NaN is false. If n = 0, then dt will be Inf. Figure 6 uses exact integer arithmetic to guarantee n executions of the for loop.

Floating point results may depend on how the code is compiled and executed. For example, w = x + y + z is executed as if it were A = x + y; w = A + z. The new anonymous variable, A, may be computed in extended precision or in the precision of x, y, and z, depending on factors often beyond control or the programmer. The results will be as accurate as they should be, but they will be exactly identical. For example, on a Pentium running Linux\(^9\), the statement

\[\text{double eps = 1e-16;}\]
\[\text{cout << 1 + eps - 1 << endl;}\]

\(^9\) This is a 64-bit Pentium 4 running Linux 2.6.18 with GCC version 4.1.2 and Intel C version 10.0.
void do_something(double tStart, double tFinal, int n)
{
    double dt = (tFinal - tStart) / n;
    double t = tStart;
    for (int i = 0; i < n; ++i) {
        // Body of loop
        t += dt;
    }
}

Figure 6: A code fragment using an integer variable to regulate the loop of Figure 5. This version also is more robust in that it runs correctly if tFinal <= tStart or if n = 0.

prints 1e-16 when compiled with the Intel C compiler, which uses 80-bit intermediate precision to compute 1 + eps, and 0 when compiled with the GNU C compiler, which uses double precision for 1 + eps.

Some compilers take what seem like small liberties with the IEEE floating point standard. This is particularly true with high levels of optimization. Therefore, the output may change when the code is recompiled with a higher level of optimization. This is possible even if the code is absolutely correct.

8.3 Plotting

Visualization tools range from simple plots and graphs to more sophisticated surface and contour plots, to interactive graphics and movies. They make it possible to explore and understand data more reliably and faster than simply looking at lists of numbers. We discuss only simple graphs here, with future software sections discussing other tools. Here are some things to keep in mind.

Learn your system and your options. Find out what visualization tools are available or easy to get on your system. Choose a package designed for scientific visualization, such as MATLAB (or Gnuplot, R, Python, etc.), rather than one designed for commercial presentations such as Excel. Learn the options such as line style (dashes, thickness, color, symbols), labeling, etc. Also, learn how to produce figures that are easy to read on the page as well as on the screen. Figure 7 shows a MATLAB script that sets the graphics parameters so that printed figures are easier to read.

Use scripting and other forms of automation. You will become frustrated typing several commands each time you adjust one detail of the plot. Instead, assemble the sequence of plot commands into a script.

Choose informative scales and ranges. Figure 8 shows the convergence of the fixed-point iteration from Section 4 by plotting the residual $r_i = |x_i + \log(x_i) - y|$
% Set MATLAB graphics parameters so that plots are more readable
% in print. These settings are taken from the front matter in
% L.N. Trefethen’s book ‘Spectral Methods in MATLAB’ (SIAM, 2000),
% which includes many beautifully produced MATLAB figures.

set(0, 'defaultaxesfontsize', 12, ...
    'defaultaxeslinewidth', .7, ...
    'defaultlinelinewidth', .8, ...
    'defaultpatchlinewidth', .7);

Figure 7: A Matlab script to set graphics parameters so that printed figures are easier to read.

against the iteration number. On a linear scale (first plot), all the residual
values after the tenth step are indistinguishable to plotting accuracy from zero.
The semi-logarithmic plot shows how big each of the iterates is. It also makes clear that the $\log(r_i)$ is nearly proportional to $i$; this is the hallmark of linear convergence, which we will discuss more in Chapter ??.

Combine curves you want to compare into a single figure. Stacks of graphs are as frustrating as arrays of numbers. You may have to scale different curves differently to bring out the relationship they have to each other. If the curves are supposed to have a certain slope, include a line with that slope. If a certain $x$ or $y$ value is important, draw a horizontal or vertical line to mark it in the figure. Use a variety of line styles to distinguish the curves. Exercise 9 illustrates some of these points.

Make plots self-documenting. Figure 8 illustrates mechanisms in Matlab for doing this. The horizontal and vertical axes are labeled with values and text. Parameters from the run, in this case $x_1$ and $y$, are embedded in the title.

9 Further reading

We were inspired to start our book with a discussion of sources of error by the book *Numerical Methods and Software* by David Kahaner, Cleve Moler, and Steve Nash [6]. Another interesting version is in *Scientific Computing* by Michael Heath [3]. A much more thorough description of error in scientific computing is Higham’s *Accuracy and Stability of Numerical Algorithms* [4]. Acton’s *Real Computing Made Real* [1] is a jeremiad against common errors in computation, replete with examples of the ways in which computations go awry – and ways those computations may be fixed.

A classic paper on floating point arithmetic, readily available online, is Goldberg’s “What Every Computer Scientist Should Know About Floating-Point Arithmetic” [2]. Michael Overton has written a nice short book *IEEE Floating
Convergence of $x_{i+1} = \log(y) - \log(x_i)$, with $x_1 = 1$, $y = 10$

Figure 8: Plots of the convergence of $x_{i+1} = \log(y) - \log(x_i)$ to a fixed point on linear and logarithmic scales.
Solve the equation $x\cdot \exp(x) = y$ by the fixed-point iteration

$$x(i+1) = \log(y) - \log(x(i));$$

and plot the convergence of the residual $|x+\log(x)-\log(y)|$ to zero.

**Inputs:**
- $x_1$ - Initial guess (default: 1)
- $y$ - Right side (default: 10)
- $n$ - Maximum number of iterations (default: 70)
- $fname$ - Name of an eps file for printing the convergence plot.

```matlab
function sec2_8b(x, y, n, fname)
    % Set default parameter values
    if nargin < 1, x = 1; end
    if nargin < 2, y = 10; end
    if nargin < 3, n = 70; end

    % Compute the iterates
    for i = 1:n-1
        x(i+1) = log(y) - log(x(i));
    end

    % Plot the residual error vs iteration number on a log scale
    f = x + log(x) - log(y);
    semilogy(abs(f), 'x-');

    % Label the x and y axes
    xlabel('i');
    ylabel('Residual $|x_i + \log(x_i) - \log(y)|$');

    % Add a title. The sprintf command lets us format the string.
    title(sprintf(['Convergence of $x_{i+1} = \log(y)-\log(x_i)$, with $x_1 = %g, y = %g$'], x(1), y));
    grid; % A grid makes the plot easier to read

    % If a filename is provided, print to that file (sized for book)
    if nargin ==4
        set(gcf, 'PaperPosition', [0 0 6 3]);
        print('-deps', fname);
    end
end
```

Figure 9: MATLAB code to plot convergence of the iteration from Section 4.
Point Arithmetic [9] that goes into more detail.

There are many good books on software design, among which we recommend *The Pragmatic Programmer* by Hunt and Thomas [5] and *The Practice of Programming* by Kernighan and Pike [7]. There are far fewer books that specifically address numerical software design, but one nice exception is *Writing Scientific Software* by Oliveira and Stewart [8].

10 Exercises

1. It is common to think of $\pi^2 = 9.87$ as approximately ten. What are the absolute and relative errors in this approximation?

2. If $x$ and $y$ have type double, and $(\text{fabs}(x - y) >= 10)$ evaluates to TRUE, does that mean that $y$ is not a good approximation to $x$ in the sense of relative error?

3. Show that $f_{jk} = \sin(x_0 + (j - k)\pi/3)$ satisfies the recurrence relation

$$ f_{j,k+1} = f_{j,k} - f_{j+1,k}. \tag{15} $$

We view this as a formula that computes the $f$ values on level $k+1$ from the $f$ values on level $k$. Let $\hat{f}_{jk}$ for $k \geq 0$ be the floating point numbers that come from implementing $f_{j0} = \sin(x_0 + j\pi/3)$ and (15) (for $k > 0$) in double precision floating point. If $|\hat{f}_{jk} - f_{jk}| \leq \epsilon$ for all $j$, show that $|\hat{f}_{j,k+1} - f_{j,k+1}| \leq 2\epsilon$ for all $j$. Thus, if the level $k$ values are very accurate, then the level $k+1$ values still are pretty good.

Write a program (C/C++ or MATLAB) that computes $e_k = \hat{f}_{0k} - f_{0k}$ for $1 \leq k \leq 60$ and $x_0 = 1$. Note that $f_{0n}$, a single number on level $n$, depends on $f_{0,n-1}$ and $f_{1,n-1}$, two numbers on level $n-1$, and so on down to $n$ numbers on level 0. Print the $e_k$ and see whether they grow monotonically. Plot the $e_k$ on a linear scale and see that the numbers seem to go bad suddenly at around $k = 50$. Plot the $e_k$ on a log scale. For comparison, include a straight line that would represent the error if it were exactly to double each time.

4. What are the possible values of $k$ after the for loop is finished?

```c
float x = 100.0*rand() + 2;
int n = 20, k = 0;
float dy = x/n;
for (float y = 0; y < x; y += dy)
    k++;
/* Body does not change x, y, or dy */
```

5. We wish to evaluate the function $f(x)$ for $x$ values around $10^{-3}$. We expect $f$ to be about $10^5$ and $f'$ to be about $10^{10}$. Is the problem too ill-conditioned for single precision? For double precision?
6. Use the fact that the floating point sum $x + y$ has relative error $\epsilon < \epsilon_{\text{mach}}$ to show that the absolute error in the sum $S$ computed below is no worse than $(n - 1)\epsilon_{\text{mach}} \sum_{k=0}^{n-1} |x_i|:

```c
double compute_sum(double x[], int n)
{
    double S = 0;
    for (int i = 0; i < n; ++i)
        S += x[i];
    return S;
}
```

7. Starting with the declarations

```c
float x, y, z, w;
const float oneThird = 1/ (float) 3;
const float oneHalf = 1/ (float) 2;
// const means these never are reassigned
we do lots of arithmetic on the variables x, y, z, w. In each case below, determine whether the two arithmetic expressions result in the same floating point number (down to the last bit) as long as no NaN or inf values or denormalized numbers are produced.
```

(a) \[(x \times y) + (z - w)
      (z - w) + (y \times x)\]

(b) \[(x + y) + z
      x + (y + z)\]

(c) \[x \times \text{oneHalf} + y \times \text{oneHalf}
      (x + y) \times \text{oneHalf}\]

(d) \[x \times \text{oneThird} + y \times \text{oneThird}
      (x + y) \times \text{oneThird}\]

8. The Fibonacci numbers, $f_k$, are defined by $f_0 = 1$, $f_1 = 1$, and

$$f_{k+1} = f_k + f_{k-1}$$

for any integer $k > 1$. A small perturbation of them, the pib numbers ("p" instead of "f" to indicate a perturbation), $p_k$, are defined by $p_0 = 1$, $p_1 = 1$, and

$$p_{k+1} = c \cdot p_k + p_{k-1}$$

for any integer $k > 1$, where $c = 1 + \sqrt{3}/100$. 

(a) Plot the $f_n$ and $p_n$ in one together on a log scale plot. On the plot, mark $1/\epsilon_{\text{mach}}$ for single and double precision arithmetic. This can be useful in answering the questions below.

(b) Rewrite (16) to express $f_{k-1}$ in terms of $f_k$ and $f_{k+1}$. Use the computed $f_n$ and $f_{n-1}$ to recompute $f_k$ for $k = n-2, n-3, \ldots, 0$. Make a plot of the difference between the original $f_0 = 1$ and the recomputed $\hat{f}_0$ as a function of $n$. What $n$ values result in no accuracy for the recomputed $f_0$? How do the results in single and double precision differ?

(c) Repeat b. for the pib numbers. Comment on the striking difference in the way precision is lost in these two cases. Which is more typical? Extra credit: predict the order of magnitude of the error in recomputing $p_0$ using what you may know about recurrence relations and what you should know about computer arithmetic.

9. The binomial coefficients, $a_{n,k}$, are defined by

$$a_{n,k} = \begin{pmatrix} n \\ k \end{pmatrix} = \frac{n!}{k!(n-k)!}$$

To compute the $a_{n,k}$, for a given $n$, start with $a_{n,0} = 1$ and then use the recurrence relation $a_{n,k+1} = \frac{n-k}{k+1} a_{n,k}$.

(a) For a range of $n$ values, compute the $a_{n,k}$ this way, noting the largest $a_{n,k}$ and the accuracy with which $a_{n,n} = 1$ is computed. Do this in single and double precision. Why is roundoff not a problem here as it was in problem 8? Find $n$ values for which $\hat{a}_{n,n} \approx 1$ in double precision but not in single precision. How is this possible, given that roundoff is not a problem?

(b) Use the algorithm of part (a) to compute

$$E(k) = \frac{1}{2^n} \sum_{k=0}^{n} ka_{n,k} = \frac{n}{2}.$$  

Write a program without any safeguards against overflow or zero divide (this time only)! Show (both in single and double precision) that the computed answer has high accuracy as long as the intermediate results are within the range of floating point numbers. As with (a), explain how the computer gets an accurate, small, answer when the intermediate numbers have such a wide range of values. Why is cancellation not a problem? Note the advantage of a wider range of values: we can compute $E(k)$ for much larger $n$ in double precision. Print $E(k)$ as computed by (17) and $M_n = \max_k a_{n,k}$. For large $n$, one should be $\inf$ and the other NaN. Why?

---

10. One of the purposes of the IEEE floating point standard was to allow a program with overflow or zero divide to run and print results.
(c) For fairly large $n$, plot $a_{n,k}/M_n$ as a function of $k$ for a range of $k$ chosen to illuminate the interesting “bell shaped” behavior of the $a_{n,k}$ near $k = n/2$. Combine the curves for $n = 10$, $n = 20$, and $n = 50$ in a single plot. Choose the three $k$ ranges so that the curves are close to each other. Choose different line styles for the three curves.
References


