

Chapter 2

Sources of Error

In scientific computing, we never expect to get the exact answer. Inexactness is practically the definition of scientific computing. Getting the exact answer, generally with integers or rational numbers, is *symbolic computing*, an interesting but distinct subject. Suppose we are trying to compute the number A . The computer will produce an approximation, which we call \hat{A} . This \hat{A} may agree with A to 16 decimal places, but the identity $A = \hat{A}$ (almost) never is true in the mathematical sense, if only because the computer does not have an exact representation for A . For example, if we need to find x that satisfies the equation $x^2 - 175 = 0$, we might get 13 or 13.22876, depending on the computational method, but $\sqrt{175}$ cannot be represented exactly as a floating point number.

Four primary sources of error are: (i) *roundoff error*, (ii) *truncation error*, (iii) *termination of iterations*, and (iv) *statistical error* in Monte Carlo. We will estimate the sizes of these errors, either *a priori* from what we know in advance about the solution, or *a posteriori* from the computed (approximate) solutions themselves. Software development requires distinguishing these errors from those caused by outright bugs. In fact, the bug may not be that a formula is wrong in a mathematical sense, but that an approximation is not accurate enough. This chapter discusses floating point computer arithmetic and the IEEE floating point standard. The others are treated later.

Scientific computing is shaped by the fact that nothing is exact. A mathematical formula that would give the exact answer with exact inputs might not be robust enough to give an approximate answer with (inevitably) approximate inputs. Individual errors that were small at the source might combine and grow in the steps of a long computation. Such a method is *unstable*. A problem is *ill conditioned* if any computational method for it is unstable. *Stability* theory, which is modeling and analysis of error growth, is an important part of scientific computing.

2.1 Relative error, absolute error, and cancellation

The *absolute error* in approximating A by \hat{A} is $e = \hat{A} - A$. The *relative error*, which is $\epsilon = e/A$, is usually more meaningful. These definitions may be restated as

$$\hat{A} = A + e \quad (\text{absolute error}), \quad \hat{A} = A \cdot (1 + \epsilon) \quad (\text{relative error}). \quad (2.1)$$

For example, the absolute error in approximating $A = \sqrt{175}$ by $\hat{A} = 13$ is $e = 13.22876 \dots - 13 \approx .23$. The corresponding relative error is $e/A \approx .23/13.2 \approx .017 < 2\%$. Saying that the error is less than 2% is probably more informative than saying that the error is less than $.25 = 1/4$.

Relative error is a *dimensionless* measure of error. In practical situations, the desired A probably has units, such as seconds, meters, etc. If A is a length measured in meters, knowing $e \approx .23$ does not tell you whether e is large or small. If the correct length is half a meter, then $.23$ is a large error. If the

correct length in meters is $13.22876\dots$, then \hat{A} is off by less than 2%. If we switch to centimeters the error becomes 22.9. This may seem larger, but it still is less than 2% of the exact length, $1,322.876\dots$ (in centimeters).

We often describe the accuracy of an approximation or measurement by saying how many decimal digits are correct. For example, Avogadro's number (the number of molecules in one mole) 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 we believe the 0 is correct, that the true Avogadro's number is closer to 6×10^{23} than to 6.1×10^{23} or 5.9×10^{23} . With three digits the number is $N_0 \approx 6.02 \times 10^{23}$. In an absolute sense, the difference between $N_0 \approx 6 \times 10^{23}$ and $N_0 \approx 6.02 \times 10^{23}$ is 2×10^{21} molecules per mole, which may seem like a lot, but the relative error is about a third of one percent.

While relative error is more useful than absolute error, it also is more problematic. Relative error can grow through *cancellation*. For example, suppose we know $A = B - C$ and we have evaluated B and C to three decimal digits of accuracy. If the first two digits of B and C agree, then they *cancel* in the subtraction, leaving only one correct digit in A . If, say, $B \approx \hat{B} = 2.38 \times 10^5$ and $C \approx \hat{C} = 2.33 \times 10^5$, then $A \approx \hat{A} = 5 \times 10^3$. This \hat{A} is probably off by more than 10% even though \hat{B} and \hat{C} had relative error less than 1%. *Catastrophic cancellation* is losing many digits in one subtraction. More subtle and more common is an accumulation of less dramatic cancellations over a series of steps.

2.2 Computer arithmetic

Error from inexact computer floating point arithmetic is called *roundoff error*. Roundoff error occurs in most floating point operations. Some computations involve no other approximations. For example, solving systems of linear equations using Gaussian elimination would give the exact answer in *exact arithmetic* (all computations performed exactly). Even these computations can be unstable and give wrong answers. Being exactly right in exact arithmetic does not imply being approximately right in floating point arithmetic.

Floating point arithmetic on modern computers is governed by the *IEEE floating point standard*. Following the standard, a floating point operation normally has relative error less than the *machine precision*, but of the same order of magnitude. The machine precision is $\epsilon_{\text{mach}} \approx 6 \cdot 10^{-8}$ for *single precision* (data type `float` in C), and $\epsilon_{\text{mach}} = 2^{-53} \approx 10^{-16}$ for *double precision* (data type `double` in C). Let $A = B \circ C$, with \circ standing for one of the arithmetic operations: addition ($A = B + C$), subtraction, multiplication, or division. With the same B and C , the computer will produce \hat{A} with relative error (2.1) that normally satisfies $|\epsilon| \leq \epsilon_{\text{mach}}$.

2.2.1 Introducing the standard

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 four 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.
4. To give the programmer control over exception handling.

The standard specifies exactly how numbers are represented in hardware. The most basic unit of information that a computer stores is a *bit*, a variable whose value may be either 0 or 1. Bits are organized into 32 bit or 64 bit *words*, or *bit strings*. The number of 32 bit words is¹ $2^{32} = 2^2 \cdot 2^{30} \approx 4 \times (10^3)^3 = 4$ billion. A typical computer should take well under a minute to list all of them. A computer running at 1GHz in theory can perform one billion operations per second, though that may not be achieved in practice. The number of 64 bit words is about $1.6 \cdot 10^{19}$, which is too many to be listed in a year. A 32 bit floating point number is called *single precision* and has data type `float` in C/++. A 64 bit floating point number is called *double precision* and has data type `double`.

C/C++ also has data types `int` (for 32 bits) and `longint` (for 64 bits) that represent integers. Integer, or *fixed point* arithmetic, is very simple. With 32 bit integers, the $2^{32} \approx 4 \cdot 10^9$ distinct words represent that many consecutive integers, filling the range from about $-2 \cdot 10^9$ to about $2 \cdot 10^9$. Addition, subtraction, and multiplication are done exactly whenever the answer is within this range. The result is unpredictable when the answer is out of range (*overflow*). Results of integer division are rounded down to the nearest integer below the answer.

2.2.2 Representation of numbers, arithmetic operations

For scientific computing, integer arithmetic has two drawbacks. One is that there is no representation for numbers that are not integers. Also important is the small range of values. The number of dollars in the US national debt, several trillion (10^{12}), cannot be represented as a 32 bit integer but is easy to approximate in 32 bit floating point.

The standard assigns a real number value to each single precision or double precision bit string. On a calculator display, the expression:

$$-.2491E - 5$$

means $-2.491 \cdot 10^{-6}$. This expression consists of a sign bit, $s = -$, a mantissa, $m = 2491$ and an exponent, $e = -5$. The expression $s.mEe$ corresponds to the number $s \cdot m \cdot 10^e$. Scientists like to put the first digit of the mantissa on the left of the decimal point ($-2.491 \cdot 10^{-6}$) while calculators put the whole thing on the

¹We use the approximation $2^{10} = 1024 \approx 10^3$.

right ($-.2491 \cdot 10^{-5}$). In base 2 (binary) arithmetic, the scientists' convention saves a bit, see below.

When the standard interprets a 32 bit word, the first bit is the *sign bit*, $s = \pm$. The next 8 bits form the *exponent*², e , and the remaining 23 bits determine the form the *fraction*, f . There are two possible signs, $2^8 = 256$ possible values of e (ranging from 0 to 255), and $2^{23} \approx 8$ million possible fractions. Normally a floating point number has the value

$$A = \pm 2^{e-127} \cdot (1.f)_2 \quad , \quad (2.2)$$

where f is base 2 and the notation $(1.f)_2$ means that the expression $1.f$ is interpreted in base 2. Note that the mantissa is $1.f$ rather than just the fractional part, f . Any number (except 0) can be normalized so that its base 2 mantissa has the form $1.f$. There is no need to store the "1." explicitly, which saves one bit.

For example, the number $2.752 \cdot 10^3 = 2572$ can be written

$$\begin{aligned} 2752 &= 2^{11} + 2^9 + 2^7 + 2^6 \\ &= 2^{11} \cdot (1 + 2^{-2} + 2^{-4} + 2^{-5}) \\ &= 2^{11} \cdot (1 + (.01)_2 + (.0001)_2 + (.00001)_2) \\ &= 2^{11} \cdot (1.01011)_2 \quad . \end{aligned}$$

Altogether, we have, using $11 = (1011)_2$,

$$2752 = +(1.01011)_2^{(1011)_2} \quad .$$

Thus, we have sign $s = +$. The exponent is $e - 127 = 11$ so that $e = 138 = (10001010)_2$. The fraction is $f = (010110000000000000000000)_2$. The entire 32 bit string corresponding to $2.752 \cdot 10^3$ then is:

$$\underbrace{1}_s \underbrace{10001010101011000000000000000000}_e \underbrace{010110000000000000000000}_f \quad .$$

For arithmetic operations, the standard mandates the rule: *the exact answer, correctly rounded*. For example, suppose x , y , and z are computer variables of type `float`, and the computer executes the statement $x = y / z$;. Let B and C be the numbers that correspond to the 32 bit strings y and z using the standard (2.2). A number that can be represented exactly in form (2.2) using 32 bits is a (32 bit) *floating point number*. Clearly B and C are floating point numbers, but the exact quotient, $A = B/C$, probably is not. *Correct rounding* means finding the floating point number \hat{A} closest³ to A . The computer is supposed to set the bit string x equal to the bit string representing \hat{A} . For exceptions to this rule, see below.

²This a slight misnomer; the actual exponent is $e - 127$ (in single precision) exponent.

³Ties can happen. The accuracy of IEEE floating point arithmetic does not depend on how ties are resolved.

The exact answer correctly rounded rule implies that the only error in floating point arithmetic comes from rounding the exact answer, A , to the nearest floating point number, \hat{A} . This rounding error is determined by the distance between floating point numbers. The greatest rounding is when A is half way between neighboring floating point numbers, B_- and B_+ . For a floating point number of the form $B_- = (1.f_-)_2 \cdot 2^p$, the next larger floating point number is usually $B_+ = (1.f_+)_2 \cdot 2^p$, where we get f_+ from f_- by adding the smallest possible fraction, which is 2^{-23} for 23 bit single precision fractions. The *relative* size of the gap between B_- and B_+ is, after some algebra,

$$\gamma = \frac{B_+ - B_-}{B_-} = \frac{(1.f_+)_2 - (1.f_-)_2}{(1.f_-)_2} = \frac{2^{-23}}{(1.f_-)_2}.$$

The largest γ is given by the smallest denominator, which is $(1.0 \cdots 0)_2 = 1$, which gives $\gamma_{max} = 2^{-23}$. The largest rounding error is half the gap size, which gives the single precision machine precision $\epsilon_{mach} = 2^{-24}$ stated above.

The 64 bit double precision floating point format allocates one bit for the sign, 11 bits for the exponent, and the remaining 52 bits for the fraction. Therefore its floating point precision is given by $\epsilon_{mach} = 2^{-53}$. Double precision arithmetic gives roughly 16 decimal digits of accuracy instead of 7 for single precision. There are 2^{11} possible exponents in double precision, ranging from 1023 to -1022 . The largest double precision number is of the order of $2^{1023} \approx 10^{307}$. The largest single precision number is about $2^{126} \approx 10^{38}$. Not only is double precision arithmetic more accurate than single precision, but the range of numbers is far greater.

2.2.3 Exceptions

The extreme exponents, $e = 0$ and $e = 255$ in single precision ($e = 0$ and $e = 2^{11} - 1 = 2047$ in double), are not interpreted using (2.2). Instead, they have carefully engineered interpretations that make the IEEE standard distinctive. Numbers with $e = 0$ are *denormalized* and have the value

$$A = \pm 0.f \cdot 2^{-126} \quad (\text{single precision}), \quad A = \pm 0.f \cdot 2^{-1022} \quad (\text{double}).$$

This feature is called *gradual underflow*. *Underflow* is the situation in which the result of an operation is not zero but is closer to zero than any normalized floating point number. In single precision, the smallest normalized positive floating point number is $A = (1.0 \cdots 0)_2 \cdot 2^{-126}$. The nearest floating point number in the positive direction is $B_+ = (1.0 \cdots 01)_2 \cdot 2^{-126}$. The nearest floating point number in the negative direction is the denormalized number $B_- = (0.1 \cdots 11)_2 \cdot 2^{-126}$. The gap between A and B_+ and the gap between B_- and A both are $(0.0 \cdots 01)_2 \cdot 2^{-126} = 2^{-126-23} = 2^{-149}$. Without denormalized numbers, A would have a gap of size 2^{-149} on the right and 2^{-126} (the space between 0 and A) on the left: the left gap would be $2^{23} \approx 4$ billion times larger than the gap on the right. 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.

The other extreme case, $e = 255$ in single precision, has two subcases, `inf` (for *infinity*) if $f = 0$ and `NaN` (for *Not a Number*) if $f \neq 0$. The C++ statement `cout << x;` produces⁴ “`inf`” and “`NaN`” respectively. An arithmetic operation produces `inf` if the exact answer is larger than the largest floating point number, as does `1/x` if $x = \pm 0$. (Actually $1/ + 0 = +\text{inf}$ and $1/ - 0 = -\text{inf}$). Invalid operations such as `sqrt(-1.)`, `log(-4.)`, produce `NaN`. Any operation involving a `NaN` produces another `NaN`. It is planned that f will contain information about how or where in the program the `NaN` was created but this is not standardized yet. Operations with `inf` are common sense: `inf + finite = inf`, `inf/inf = NaN`, `finite/inf = 0`, `inf + inf = inf`, `inf - inf = NaN`.

A floating point arithmetic operation is an *exception* if the result is not a normalized floating point number. The standard mandates that a hardware *flag* (a binary bit of memory in the processor) should be *set* (given the value 1) when an exception occurs. There should be a separate flag for the underflow, `inf`, and `NaN` exceptions. The programmer should be able to specify what happens when an exception flag is set. Either the program execution continues without interruption or an *exception handler* procedure is called. The programmer should be able to write procedures that interface with the exception handler to find out what happened and take appropriate action. Only the most advanced and determined programmer will be able to do this. The rest of us have the worst of both: the exception handler is called, which slows the program execution but does nothing useful.

Many features of IEEE arithmetic are illustrated in Figure 2.1. Note that e^{204} gives `inf` in single precision but not in double precision because the range of values is larger in double precision. We see that `inf` and `NaN` work as promised. The main rule, “exact answer correctly rounded”, explains why adding pairs of floating point numbers is commutative: the mathematical sums are equal so they round to the same floating point number. This does not force addition to be associative, which it is not. Multiplication also is commutative but not associative. The division operator gives integer or floating point division depending on the types of the operands. Integer arithmetic *truncates* the result to the next lower integer rather than rounding it to the nearest integer.

```
// A program that explores floating point arithmetic in the IEEE
// floating point standard. The source code is SourcesOfError.C.
```

```
#include <iostream.h>
#include <math.h>
```

```
int main() {
```

⁴Microsoft, in keeping with its pattern of maximizing incompatibility, gives something different.

```

float  xs, ys, zs, ws; // Some single precision variables.
double yd;             // A double precision variable.

xs = 204.;            // Take an exponential that is out of range.
ys = exp(xs);
cout << "The single precision exponential of " << xs <<
      " is " << ys << endl;
yd = exp ( xs );    // In double precision, it is in range.
cout << "The double precision exponential of " << xs <<
      " is " << yd << endl;

zs = xs / ys;        // Divide a normal number by infinity.
cout << xs << " divided by " << ys <<
      " gives " << zs << endl;

ws = ys;            // Divide infinity by infinity.
zs = ws / ys;
cout << ws << " divided by " << ys << " gives " << zs << endl;

zs = sqrt( -1. ) ; // sqrt(-1) should be NaN.
cout << "sqrt(-1.) is " << zs << endl;

ws = xs + zs;       // Add NaN to a normal number.
cout << xs << " + " << zs << " gives " << ws << endl;

xs =      sin(1.); // Some generic single precision numbers.
ys = 100. *sin(2.);
zs = 10000.*sin(3.);
float xsPys, ysPxs, xsPzs, zsPxs; // xsPzx holds xs + zs, etc.
xsPys = xs + ys;
ysPxs = ys + xs; // Try commuting pairs.
xsPzs = xs + zs;
zsPxs = zs + xs;
if ( ( xsPys == ysPxs ) && ( xsPzs == zsPxs ) )
    cout << "Adding " << xs << " " << ys << " and " << zs <<
          " in pairs commutes." << endl;
else
    cout << "Adding " << xs << " " << ys << " and " << zs <<
          " in pairs does not commute." << endl;

float xsPysPzs, ysPzsPxs; // Test for associativity.
xsPysPzs = ( xs + ys ) + zs;
ysPzsPxs = ( ys + zs ) + xs;
if ( xsPysPzs == ysPzsPxs )

```



```

    cout << "Adding " << xs << " " << ys << " and "<< zs <<
        " is associative." << endl;
else
    cout << "Adding " << xs << " " << ys << " and "<< zs <<
        " is not associative." << endl;

int xi, yi; // Some integer variables.
xi = 9;     // Compute the quotient using integer
yi = 10;    // and floating point arithmetic.
zs = xi/yi;
ws = ( (float) xi ) / ( (float) yi ); // Convert, then divide.
cout << "Integer division of " << xi << " by " << yi <<
    " gives " << zs << ". " <<
    " Floating point gives " << ws << endl;

return(0);

}

```

Figure 2.1: A program that illustrates some of the features of arithmetic using the IEEE floating point standard.

2.3 Truncation error

Truncation error is the error in analytical approximations such as

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}. \quad (2.3)$$

This is not an exact formula, 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,

$$f(x+h) = f(x) + hf'(x) + \frac{1}{2}h^2f''(x) + \dots,$$

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 (2.3). 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.

h	.3	.01	10^{-5}	10^{-8}	10^{-10}
\widehat{f}'	6.84	5.48	5.4366	5.436564	5.436562
e_{tot}	1.40	$4.10 \cdot 10^{-2}$	$4.08 \cdot 10^{-5}$	$-5.76 \cdot 10^{-8}$	$-1.35 \cdot 10^{-6}$

Figure 2.2: Estimates of $f'(x)$ using (2.3). The error is e_{tot} , which results from truncation and roundoff error. Roundoff error is apparent only in the last two columns.

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 3 we will see that (in exact arithmetic) e_{tr} roughly is proportional to h for small h . The numbers in Figure 2.3 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.

2.4 Iterative Methods

Suppose we want to find A by solving an equation. Often it is impossible to find a formula for the solution. Instead, *iterative methods* construct a sequence of approximate solutions, A_n , for $n = 1, 2, \dots$. Hopefully, the approximations *converge* to the right answer: $A_n \rightarrow A$ as $n \rightarrow \infty$. In practice, we must stop the iteration process 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 with $xe^x = y$. There is not a 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 $x = \lim_{n \rightarrow \infty} x_n$ (if it exists), is a *fixed point* of the iteration, i.e. $x = \ln(y) - \ln(x)$, which implies $xe^x = y$. Figure 2.4 demonstrates the convergence of the iterates in this case with $y = 10$. The *initial guess* is $x_1 = 1$. After 20 iterations, we have $x_{20} \approx 1.74$. The error is $e_{20} \approx 2.3 \cdot 10^{-5}$, which might be small enough, depending on the application.

After 67 iterations the relative error is $(x_{67} - x)/x \approx 2.2 \cdot 10^{-16}/1.75 \approx 1.2 \cdot 10^{-16}$, which is only slightly larger than double precision machine precision $\epsilon_{mach} \approx 1.1 \cdot 10^{-16}$. This shows that supposedly approximate iterative methods can be as accurate as *direct* methods that would be exact in exact arithmetic. It would be a surprising fluke for even a direct method to achieve better than machine precision because even they are subject to roundoff error.

n	1	3	6	10	20	67
x_n	1	1.46	1.80	1.751	1.74555	1.745528
e_n	-.745	-.277	$5.5 \cdot 10^{-2}$	$5.9 \cdot 10^{-3}$	$2.3 \cdot 10^{-5}$	$2.2 \cdot 10^{-16}$

Figure 2.3: 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, $y = 10$.

n	10	100	10^4	10^6	10^6	10^6
\hat{A}	.603	.518	.511	.5004	.4996	.4991
error	.103	$1.8 \cdot 10^{-2}$	$1.1 \cdot 10^{-2}$	$4.4 \cdot 10^{-4}$	$-4.0 \cdot 10^{-4}$	$-8.7 \cdot 10^{-4}$

Figure 2.4: Statistical errors in a demonstration Monte Carlo computation.

2.5 Statistical error in Monte Carlo

Monte Carlo means using random numbers as a computational tool. For example, suppose⁵ $A = E[X]$, 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, \dots , 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 $\hat{A} \rightarrow A$ as $n \rightarrow \infty$. 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 2.5 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 9 for a clarification of this). The errors even with a million samples are much larger than those in the right columns of Figures 2.3 and 2.4.

2.6 Error propagation and amplification

Errors can be amplified as they *propagate* through a computation. For example, suppose the divided difference (2.3) is part of a long calculation:

$$f1 = \dots ; \quad \backslash \text{ approx of } f(x)$$

⁵ $E[X]$ is the *expected value* of X . Chapter 9 has some review of probability.

```
f2 = . . . ; \\ approx of f(x+h)
fPrimeHat = ( f2 - f1 ) / h ; \\ approx of derivative
```

It is unlikely that $f_1 = \widehat{f(x)} \approx f(x)$ is exact. Many factors may contribute to the errors $e_1 = f_1 - f(x)$ and $e_2 = f_2 - f(x+h)$. There are three contributions to the final error in f' :

$$\widehat{f}' - f' = e_r + e_{tr} + e_{pr}. \quad (2.4)$$

One is the roundoff error in evaluating $(f_2 - f_1) / h$ in floating point

$$\widehat{f}' = \frac{f_2 - f_1}{h} + e_r. \quad (2.5)$$

The truncation error in the difference quotient approximation is

$$\frac{f(x+h) - f(x)}{h} - f' = e_{tr}. \quad (2.6)$$

The *propagated error* comes from using inexact values of $f(x+h)$ and $f(x)$:

$$\frac{f_2 - f_1}{h} - \frac{f(x+h) - f(x)}{h} = \frac{e_2 - e_1}{h} = e_{pr}. \quad (2.7)$$

If we add (2.5), (2.6), and (2.7), and simplify, we get the formula (2.4).

A stage of a long calculation creates some errors and propagates errors from earlier stages, possibly with amplification. In this example, the difference quotient evaluation introduces truncation and roundoff error. Also, e_1 and e_2 represent errors generated at earlier stages when $f(x)$ and $f(x+h)$ were evaluated. These errors, in turn, could have several sources, including inaccurate x values and roundoff in the code evaluating $f(x)$. According to (2.7), the difference quotient propagates these and amplifies them by a factor of $1/h$. A typical value $h = .01$ could amplify incoming errors e_1 and e_2 by a factor of 100.

This increase in error by a large factor in one step is an example of *catastrophic cancellation*. 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 common and more subtle is *gradual error growth* over a long sequence of computational steps. Exercise 2.12 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.

Mathematical *stability theory* in scientific computing is the search for gradual error growth in computational algorithms. It focuses 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 a stability analysis, we would assume

that the subtraction is performed exactly and that the error in f_{k-1} is entirely due to errors in f_k and f_{k+1} . That is, if $\widehat{f}_k = f_k + e_k$ is the computer approximation, then the e_k satisfy the mathematical relation $e_{k-1} = e_{k+1} - e_k$, which is the *error propagation equation*. 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 happen in practically any computation.

2.7 Condition number and ill conditioned problems

The *condition number* of a computational problem measures the sensitivity of the answer to small changes in the data. If κ is the condition number, then we expect error at least $\kappa \cdot \epsilon_{mach}$, regardless of the computational 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 simplest definition of κ supposes that the answer 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 :

$$\left| \frac{\Delta A}{A} \right| \approx \kappa \left| \frac{\Delta x}{x} \right|. \quad (2.8)$$

Any algorithm that computes $A(x)$ must round x to the nearest floating point number \widehat{x} , which creates a relative error $\Delta x/x = (\widehat{x} - x)/x \sim \epsilon_{mach}$. If the rest of the computation were done exactly, the computed answer would be $\widehat{A}(x) = A(\widehat{x})$ and the relative error would be (using (2.8))

$$\left| \frac{\widehat{A}(x) - A(x)}{A(x)} \right| = \left| \frac{A(\widehat{x}) - A(x)}{A(x)} \right| \approx \kappa \left| \frac{\Delta x}{x} \right| \sim \kappa \epsilon_{mach}. \quad (2.9)$$

If A is a differentiable function of x with derivative $A'(x)$, then, for small Δx , $\Delta A \approx A'(x)\Delta x$. With a little algebra, this gives (2.8) with

$$\kappa = \left| A'(x) \cdot \frac{x}{A(x)} \right|. \quad (2.10)$$

This analysis argues that any computational algorithm for an ill conditioned problem must be unstable. Even if $A(x)$ is evaluated exactly, relative errors in the input of size ϵ are amplified by a factor of κ . The formulas (2.9) and (2.10) represent an absolute lower bound for the accuracy of any computational algorithm. An ill conditioned problem is not going to be solved accurately, period.

The formula (2.10) gives a dimensionless κ 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 (2.8) are dimensionless with dimensionless κ .

As an example consider the problem of evaluating $A(x) = R \sin(x)$. The condition number formula (2.10) gives

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

Note that the problem remains well conditioned (κ is not large) as $x \rightarrow 0$, 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 $x \rightarrow \pi$, 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 $x = (x_1, \dots, x_n)$. 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 (2.8) as saying that $|\Delta A/A| \approx \kappa \epsilon$ for $|\Delta x/x| = \epsilon$, a worst case multicomponent generalization could be

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

We seek the worst case⁶ Δx . For small ϵ 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 (2.10):

$$\kappa = \sum_{k=1}^n \left| \frac{\partial A}{\partial x_k} \cdot \frac{x_k}{A} \right|. \quad (2.11)$$

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.

2.8 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

⁶As with rounding, typical errors tend to be order of the worst case error.

```

double tStart, tFinal, t, dt;
int    n;
tStart = . . . ; // Some code that determines the start
tFinal = . . . ; // and ending time and the number of
n      = . . . ; // equal size steps.
dt     = ( tFinal - tStart ) / n; // The size of each step.
for ( t = tStart, t < tFinal, t+= dt )
    { . . . } // Body of the loop does not assign t.

```

Figure 2.5: A code fragment illustrating a pitfall of using a floating point variable to regulate a while loop.

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. There is a set of principles for scientific programming that goes beyond those for general programming (modular design, commenting, etc.). Projects are handsomely rewarded for extra efforts and care taken to do the software “right”.

2.8.1 Floating point numbers are (almost) never equal

Because of inexact floating point arithmetic, two numbers that should be equal in exact arithmetic often are not equal in the computer. In general, an equality test between two variables of type `float` or `double` is a mistake. A striking illustration of this can happen with Intel processor chips, where variables of type `double` are stored on the chip in 80 bit registers but in memory with the standard 64. Moving a variable from the register to memory loses the extra bits. Thus, a program can execute the instruction `y1 = y2`; and then do not reassign either `y1` or `y2`, but later `(y1 == y2)` may evaluate to `false` because `y1` but not `y2` was copied from register to memory.

A common mistake in this regard is to use floating point comparisons to regulate a loop. Figure 2.5 illustrates this. In exact arithmetic this would give the desired n iterations. Because of floating point arithmetic, after the n^{th} iteration, the variable `t` may be equal to `tFinal` but is much more likely to be above or below due to roundoff error. It is impossible to predict which way the roundoff error will go. We do not know whether this code will execute the `while` loop body n or $n + 1$ times. Figure 2.6 uses exact integer arithmetic to guarantee n executions of the `for` loop body.

```

double tStart, tFinal, t, dt;
int    n,      , i;
tStart = . . . ; // Some code that determines the start
tFinal = . . . ; // and ending time and the number of
n      = . . . ; // equal size steps.
dt     = ( tFinal - tStart ) / n; // The size of each step.
for ( i = 0, i < n, i++ )
    { t = tStart + i*dt; // In case the value of t is needed
      . . . }           // in the loop body.

```

Figure 2.6: A code fragment using an integer variable to regulate the loop of Figure 2.5.

2.8.2 Plotting

Careful visualization is a key step in understanding any data. Pictures can be more informative than tables of numbers. Explore and understand the data by plotting it in various ways. There are many ways to visualize data, simple graphs, surface plots, contour and color plots, movies, etc. We discuss only simple graphs here. 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, rather than one designed for commercial presentations such as Excel. Learn the options such as line style (dashes, thickness, color, symbols), labeling, etc.

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.

Frame the plot. The Matlab `plot` function with values in the range from 1.2 to 1.22 will use a vertical scale from 0 to 2 and plot the data as a nearly horizontal line, unless you tell it otherwise. Figure 2.7, presents the first 70 Fibonacci numbers. The Fibonacci numbers, f_i , are defined by $f_0 = f_1 = 1$, and $f_{i+1} = f_i + f_{i-1}$, for $i \geq 1$. On the linear scale, f_1 through f_{57} sit on the horizontal axis, indistinguishable to *plotting accuracy* from zero. The log plot shows how big each of the 70 numbers is. It also makes it clear that $\log(f_i)$ is nearly proportional to i , which implies (if $\log(f_i) \approx a + bi$, then $f_i \approx cd^i$) that the f_i are approximately exponential. If we are interested in the linear scale plot, we can edit out the useless left part of the graph by plotting only from $n = 55$ to $n = 70$.

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

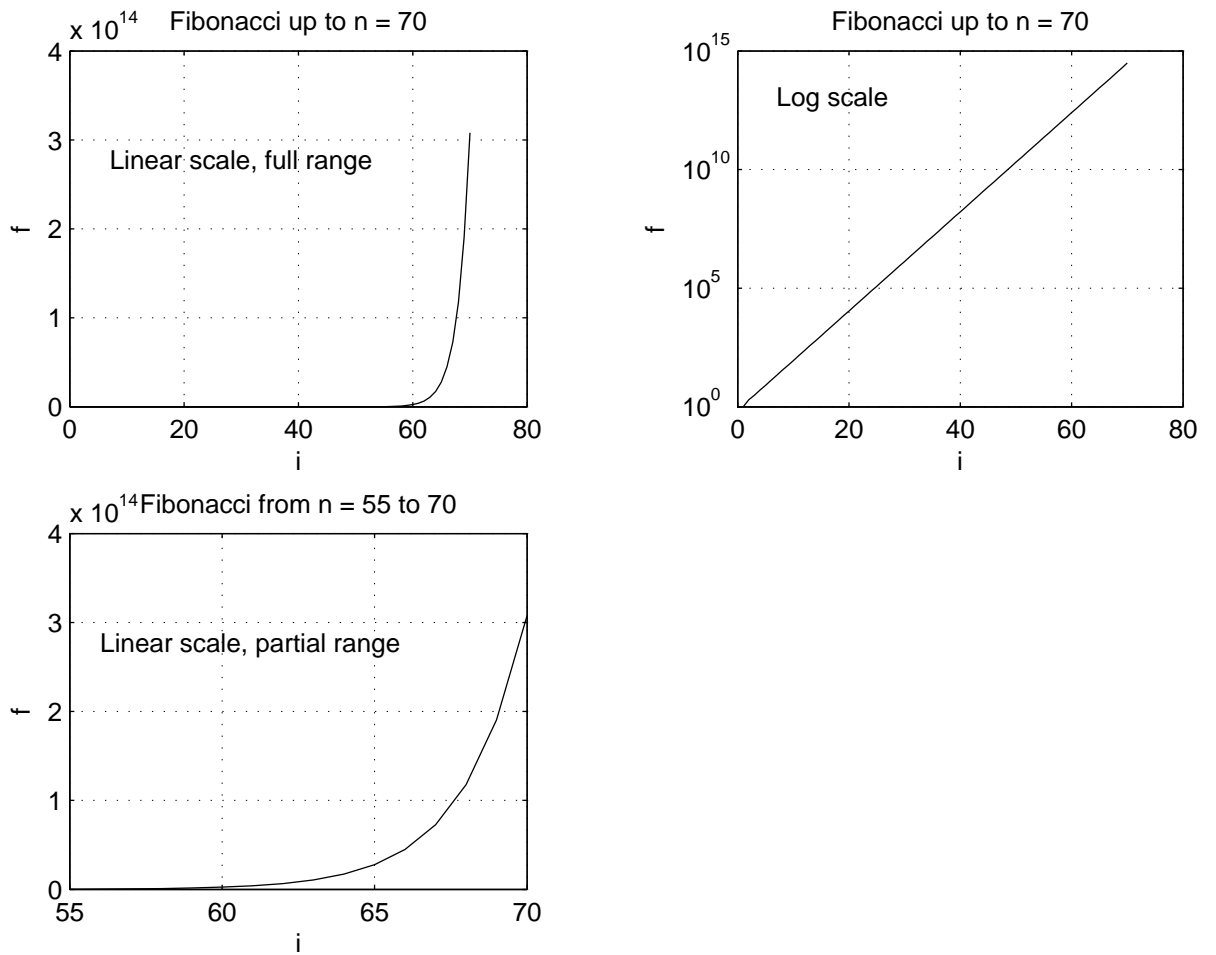


Figure 2.7: Plots of the first n Fibonacci numbers, linear scale on the left, log scale on the right

```

% Matlab code to generate and plot Fibonacci numbers.

clear f      % If you decrease the value of n, it still works.
n = 70;      % The number of Fibonacci numbers to compute.
fi = 1;      % Start with f0 = f1 = 1, as usual.
fim1 = 1;
f(1) = fi;   % Record f(1) = f1.
for i = 2:n
    fip1 = fi + fim1; % f(i+1) = f(i) + f(i-1) is the recurrence
    fim1 = fi;        % relation that defines . . .
    fi = fip1;        % the Fibonacci numbers.
    f(i) = fi;        % Record f(i) for plotting.
end

plot(f)
xlabel('i')          % The horizontal and vertical axes are
ylabel('f')          % i and f respectively.
topTitle = sprintf('Fibonacci up to n = %d',n);
                    % Put n into the title.
title(topTitle)
text(n/10, .9*f(n), 'Linear scale');
grid               % Make it easier to read values in the plot.
set ( gcf, 'PaperPosition', [.25 2.5 3.2 2.5]);
                    % Print a tiny image of the plot for the book.
print -dps FibLinear_se

```

Figure 2.8: *Matlab code to calculate and plot Fibonacci numbers.*

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 2.7 illustrates mechanisms in Matlab for doing this. The horizontal and vertical axes are labeled with values and text. In the third plot, the simpler command `plot(f(iStart:n))` would have labeled the horizontal axis from 1 to 15 (very misleading) instead of 55 to 70. Parameters from the run, in this case just n , are embedded in the title.

The Matlab script that made the plots of Figure 2.7 is in Figure 2.8. The only real parameters are n , the largest i value, and whether the plot is on a linear or log scale. Both of those are recorded in the plot. Note the convenience and clarity of not hard wiring $n = 70$. It would take just a moment to make plots up to $n = 100$.

2.9 Further reading

The idea for starting a book on computing with a discussion of sources of error comes from the book *Numerical Methods and Software* by David Kahaner, Cleve Moler, and Steve Nash. Another interesting version is in *Scientific Computing* by Michael Heath. My colleague, Michael Overton, has written a nice short book *IEEE Floating Point Arithmetic*.

2.10 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 `((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

$$f_{j,k+1} = f_{j,k} - f_{j+1,k}. \quad (2.12)$$

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 (2.12) (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?

```
float  x = 100*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. Show that in the IEEE floating point standard with any number of fraction bits, ϵ_{mach} essentially is the largest floating point number, ϵ , so that $1 + \epsilon$ gives 1 in floating point arithmetic. Whether this is exactly equivalent to the definition in the text depends on how ties are broken in rounding, but the difference between the two definitions is irrelevant (show this).
7. Starting with the declarations

```
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)

$$\begin{aligned} & (x * y) + (z - w) \\ & (z - w) + (y * x) \end{aligned}$$

(b)

$$\begin{aligned} & (x + y) + z \\ & x + (y + z) \end{aligned}$$

(c)

$$\begin{aligned} & x * oneHalf + y * oneHalf \\ & (x + y) * oneHalf \end{aligned}$$

(d)

$$\begin{aligned} & x * oneThird + y * oneThird \\ & (x + y) * oneThird \end{aligned}$$

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

$$f_{k+1} = f_k + f_{k-1} \tag{2.13}$$

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 $1/\epsilon_{mach}$ for single and double precision arithmetic. This can be useful in answering the questions below.

- (b) Rewrite (2.13) 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 $k = n-2, n-3, \dots, 0$. 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

$$a_{n,k} = \binom{n}{k} = \frac{n!}{k!(n-k)!}$$

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

$$E(k) = \frac{1}{2^n} \sum_{k=0}^n k a_{n,k} = \frac{n}{2} . \quad (2.14)$$

Write a program without any safeguards against overflow or zero divide (this time only!)⁷. 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 (2.14) and $M_n = \max_k a_{n,k}$. For large n , one should be `inf` and the other `NaN`. Why?

- (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

⁷One of the purposes of the IEEE floating point standard was to allow a program with overflow or zero divide to run and print results.

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.