Chapter 2

Floating Point Arithmetic

In this chapter we will learn how to

- utilize the ways in which a computer stores and manipulates numbers
- control the approximate nature of numerical computations
- handle special situations that arise when we attempt to evaluate irregular numerical expressions

In every day life, we perform the vast majority of our calculations using the **decimal numeral system**. This system was invented by the Hindus several thousand years before the present time and made its way to the west via the translation of the writings of the Persian astronomer and mathematician Muhammad ibn Musa al-Khwarizmi (~780–850). It uses the digits 0, 1, ..., 9, the signs + and −, as well as the decimal point (.) to represent all possible numbers.

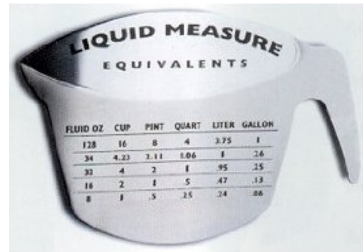
The decimal system is neither unique nor optimal for performing numerical computations. For example, multiplications and divisions are easier to perform in a system with base 12, since 12 has five integer divisors (one, two, three, four, and six) as opposed to 10 that has only three (one, two, and five). In fact, since antiquity, many civilizations have invented different numeral systems, several of which we use even today, in different aspects of everyday life (see the box in the next page). Despite its shortcoming, however, the decimal system is now the most widely used numeral system, for reasons that are believed to be intimately related to the number of fingers that we have in our hands. This fact has upset many mathematicians in the past and even led the French mathematician Blaise Pascal (1623–1662) to write "The decimal system has been established, somewhat foolishly to be sure, according to man's custom, not from a natural necessity as most people think" (as quoted in [1]).

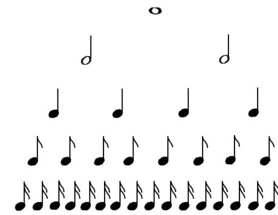
Binary System

Early computers and modern day digital electronics do not count using their fingers but rather with vacuum tubes and solid-state transistors. Both of these electronic devices have only two states: *off* and *on*. It is, therefore, natural for them to represent only the digits 0 (for off) and 1 (for on) and perform computations in the numeral system that has base 2, which is called the **binary** system. The design and arithmetic of a digital computer that is based on the binary system was explored in the 1946 classic paper by Burks, Goldstine, & von Neumann[2]. The procedures outlined in this paper form the basis of all modern computers, although

Numeral Systems in Everyday Life

Base 60: Invented by the Babylonians and passed to the ancient Greeks via the Egyptians, it became widely used in measuring positions of starts in the sky. It still survives today in the measurement of angles (a full circle has 360 degree, a degree has 60 arc minutes, an arc minute has 60 arc seconds) and of time.
Base 20: Used by the Gaul in pre-Roman France. Echoes of this system can be found in modern French, as in the word for the number 97 (quatre-vingt-dix-sept), which literally stands for four-times-twenty-plus-seventeen.
Base 12: Used extensively (often in conjunction with the decimal system) in defining the resolution of printers. Common resolutions are 360 ($=3 \times 12 \times 10$), 600 ($=3 \times 12 \times 10$), 720 ($=6 \times 12 \times 10$), 1200 ($=10 \times 12 \times 10$), or 1440 ($=12 \times 12 \times 10$) dots per inch.
Base 2: Invented by merchants in medieval England, the units of liquid measure are based on the binary system. For example, 1 gallon = 2 pottles; 1 pottle = 2 quarts; 1 quart = 2 pints, 1 pint = 2 cups, etc. The binary system is also used in defining note values (duration) in musical scores, i.e., whole note, half note, quarter note, eighth note, sixteenth note, etc.





For a more complete history of numeral systems and an extensive bibliography see [1].

a number of experimental models have been designed in the past that used the decimal or even other numeral systems.

Performing an algebraic calculation on most digital computers differs in one more fundamental way from performing the same calculation with paper and pen. A computer routinely approximates a number that has many—or even an infinite number of—digits by keeping only the most significant of those digits. This is different from a human, who will often use a particular notation to denote the correct value of the number and define new rules to perform algebraic computations with the new notation. For example, a human will write $(\sqrt{2})^2 = 2$, which is a correct result, even though $\sqrt{2}$ is a non-repeating decimal number. On the other hand, a computer that only stores the seven most significant digits of a number and truncates the rest will set $\sqrt{2} = 1.414214$ so that $(\sqrt{2})^2 = 2.000001$!

The use of the binary numeral system and the inherent approximations involved in manipulating non-repeating numbers often lead to severe errors in even the simplest numerical computations. In this chapter, we will discuss the basic rules of computer arithmetic and develop strategies to evaluate and control its limitations. For further reading, the classic book *The Art of Computer Programming. Volume II: Semi-Numerical Algorithms* by D. A. Knuth[1] provides a complete but pedagogical discussion of the subject.

2.1 The Positional Notation System

Base and Radix

In mathematics, we represent all numbers using the positional notation system. We first decide on the **base** (or **radix**) of the system to use and the notation with which to represent the various digits. For simplicity, let us assume that we pick a positive integer number b to be the base and the integers d_i with $0 \leq d_i < b$ to be the digits. Then we agree to use the notation

$$\pm(...d_3d_2d_1d_0.d_{-1}d_{-2}d_{-3}...)_b \tag{2.1}$$

to represent the number

$$\pm(... + d_3b^3 + d_2b^2 + d_1b^1 + d_0 + d_{-1}b^{-1} + d_{-2}b_{-2}...) \tag{2.2}$$

For example, when we write 1024 in the decimal notation, we imply the number $1 \times 1000 + 0 \times 100 + 2 \times 10 + 4$. In this last example, we should have formally written the number as $(1024)_{10}$, but we typically drop the subscript that denotes the base when we use the decimal system. It is worth emphasizing here that we can use any number as the base of a numeral system, including negative, irrational, or even complex numbers.

Converting a number between different numeral systems is a very simple procedure, as can be seen in the following example.

Example: Conversion between numeral systems

In order to convert a number from a numeral system with base b to the decimal system, we can simply use the definition (2.2). For example, the number $(256.4)_8$ is equal to

$$(256.4)_8 = 2 \times 8^2 + 5 \times 8^1 + 6 + 4 \times 8^{-1} = (164.5)_{10} \tag{2.3}$$

In order to perform the conversion of an integer number from the decimal system to one with base b, we start by performing a successive set of divisions of the number with the base, until the result of the final division is smaller than the value of the base. We then keep the final quotient as well as the remainder of each division, in reverse order, as the corresponding digits in the new system.

As an example, in order to convert the number $(75)_{10}$ to base 2 we perform the following divisions:

We then use the last quotient as the first digit of the number in the binary system and add the remainders of each division, in reverse order, as the remaining digits. The result is $(75)_{10} = (1001011)_2$.

If the number in the decimal system is not an integer, we convert the integer and the non-integer parts of the number separately. According to the definition (2.2), digits to the right of the decimal point correspond to successive negative powers of the base, i.e., b^{-1}, b^{-2}, etc. As a result, in order to follow the previous procedure we will need to divide successively the number with b^{-1}, which is equivalent to *multiplying* it successively by b. After each multiplication, we keep the integer part of each product as the corresponding digit in the system with base b and repeat the procedure with the non-integer part.

As an example, if we want to convert the number $(75.1)_{10}$ to base 2, we first convert the integer part as before, i.e., $(75)_{10} = (1001011)_2$ and then the non-integer part $(0.1)_{10}$ as shown below.

0.1 x 2 = 0.2 = 0.2 + 0
0.2 x 2 = 0.4 = 0.4 + 0
0.4 x 2 = 0.8 = 0.8 + 0
0.8 x 2 = 1.6 = 0.6 + 1
0.6 x 2 = 1.2 = 0.2 + 1
0.2 x 2 = 0.4 = 0.4 + 0
0.4 x 2 = 0.8 = 0.8 + 0
0.8 x 2 = 1.6 = 0.6 + 1
...

$$(0.1)_{10}=(0.00011001\ldots)_2$$

The final result is, therefore, $(75.1)_{10} = (1001011.000110011...)_2$. Note that, although the number $(0.1)_{10}$ has a finite number of digits in the decimal system, it is a repeating number in the binary system. This is a general property of the conversion of numbers between different numeral systems with often catastrophic results, as we will discuss later in this chapter.

Hexadecimal System

When we represent a number in a system with a base that is less than or equal to 10, it is customary that we use the usual Arabic numerals $0, ..., 9$ as the digits. For numbers in systems with larger bases, however, the usual numerals do not suffice and we have to incorporate additional characters for digits larger than nine. It is again customary that we use capital letters of the Latin alphabet for these high digits. For example, the number $(10)_{10}$ in base 16 is $(A)_{16}$, the number $(11)_{10}$ is $(B)_{16}$, the number $(16)_{10}$ is $(F)_{16}$, and the number $(17)_{10}$ is $(11)_{16}$.

The numeral system with base 16, or **hexadecimal** system, has a special relation to the binary system, because its base is equal to 2^4. This property makes it perhaps the second most commonly used numeral system in computational algorithms for a number of reasons. First, every four digits of a number in a binary system correspond to a single digit in the hexadecimal system. The hexadecimal system, therefore, offers a quick shorthand for numbers in the binary system, which tend to have a large number of digits. Consider the number $(75)_{10} = (01001011)_2$ of the last example, to which we have simply added the leading zero in the binary representation. The last four binary digits of this number will correspond to the last digit in the hexadecimal notation, i.e., $(1011)_2 = (B)_{16}$. The leading four binary digits of the number will correspond to the next digit in the hexadecimal system, i.e., $(0100)_2 = (8)_{16}$. As a result, we can write the long binary number $(01001011)_2$ in the shorthand $(8B)_{16}$. Second, the hexadecimal representation of a

number preserves the information related to the number of digits required to store the number in the binary system. For example, the number $(8B)_{16}$ requires at most (2 hex digits) $\times$ (4 binary per hex digits) = 8 binary digits to be stored. Finally, if the binary representation of a number has a finite number of digits to the right of the decimal point so will its hexadecimal representation and vice versa. This last property is important for the approximate representation of real numbers by a computer, to which we now turn.

2.2 Floating point arithmetic

Positional notation systems allow us, in principle, to represent any real number, from the smallest to the largest. In many cases, however, this is impractical, if a number requires a very large (or even an infinite) number of digits to be represented. For example, the constant $\pi = 3.141592...$ has a non-repeating representation in all numeral systems and requires and infinite number of digits to the right of the decimal point.

Software packages that perform calculations using symbolic manipulations (such as Mathematica and Matlab) overcome this problem by explicitly setting rules for performing algebraic calculations with real numbers. On the other hand, numerical computations that do not employ symbolic manipulation require a scheme for the approximate representation of all real numbers. Floating point arithmetic is such an approximate scheme in which real numbers are encoded and manipulated using a series of integer numbers. It is the computer equivalent to scientific notation.

The easiest way to describe the floating point representation of a number is through an example. Consider the result of the mathematical expression $e^6 \simeq 403.42879...$ In order to express this number in what is often called normalized floating point notation, we first write it in scientific notation as $e^6 = 4.0342879... \times 10^2$ such that the number without the exponent is always smaller than the base. We then decide on the number of **significant digits** that we will retain and keep *(i)* the sign of the number, *(ii)* the **exponent** in the power of ten, and *(iii)* the string of significant digits, which is called the **significand** or **mantissa**. The floating point representation of e^6 with 4 significant digits is

`e`^{6} `= (+,2,4034)`

whereas its representation with 8 significant digits is

`e`^{6} `= (+,2,40342879)`

The above example, of course, was written in the decimal system, but the same idea can be applied in any other numeral system such as the binary system used by a digital computer. In the latter case, the exponent corresponds to a power of 2, which is the base of the binary system.

In 1985, many aspects of floating point arithmetic, such as the encoding and precision with which numbers are represented, their rounding, and the handling of special situations were standardized by the IEEE 754 Standard[3]. This standard was extended in 1987 to include floating point arithmetic in the decimal system and was further revised in 2008.

Single Precision

According to the IEEE 754 standard, a floating point number of **single precision** requires 32 binary digits (or **bits**) for its storage. (Note that in the 2008 revision of the IEEE 754 Standard, this format is called `binary32`.) The first bit represents the sign of the number. It is zero for a positive number and one for a negative number. The following 8 bits store (in binary format) the exponent and

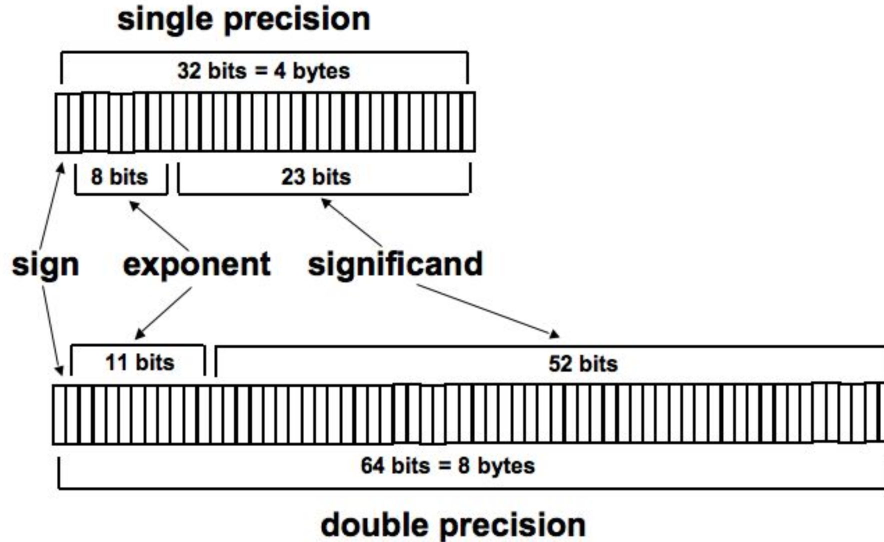


Figure 2.1: The storage format of a 32-bit single precision number (or binary32) and of a 64-bit double precision number (or binary64).

the remaining 23 bits store the digits of the significand.

The exponent may, in principle, be positive or negative. In order to avoid defining one bit as the sign of the exponent, the IEEE 754 requires that the number 127 is added to the value of the real exponent. The smallest number that can be stored in 8 bits is zero and the largest number is $(11111111)_2 = (255)_{10}$. As a result, the smallest actual exponent that can be stored is $0 - 127 = -127$, whereas the largest is $255 - 127 = 128$. This implies that the smallest and largest numbers of either sign that can be represented with single precision are approximately $2^{128} \simeq 3 \times 10^{38}$ and $2^{-127} \simeq 6 \times 10^{-39}$, respectively. These correspond to variables of type `float` in the C programming language.

Double Precision

A **double precision** number requires 64 bits for its storage. (In the 2008 revision of the IEEE 758 Standard, this format is called `binary64`.) As in the previous case, the first bit holds its sign, the following 11 bits hold its exponent, from which the number 1023 has be subtracted, and the remaining 52 bits hold the digits of the significand. The smallest number that can be stored in 11 bits is again zero, but the largest number is $2^{11} - 1 = 2047$. As a result, the largest and smallest numbers that can be represented with double precision are approximately $2^{2047-1023} \simeq$ and $2^{0-1023} \simeq$, respectively. These correspond to variables of type `double` in the C programming language.

In most situations, the C compiler deals automatically with the conversion between variable types when we perform a calculation that mixes variables of types `float` and `double` (recall, though, the example at the top of page 1-10 for an exception). Understanding, however, the storage requirements and inherent limitations of floating point arithmetic is crucial in developing accurate and efficient algorithms for solving physical problems. In extreme cases, lack of understanding of floating point arithmetic may have costly or even deadly consequences (see the boxes in the following pages). In the remaining of this chapter, we will discuss the accuracy of floating point arithmetic and leave the issues related to its efficient use for Chapter 5.

Data type conversion errors can be very costly

Ariane 5 is the primary launch vehicle of the European Space Agency (ESA). that operates from the Guiana Space Center near Kourou in the French Guiana. Its first successful operational flight took place in December 1999, when it carried to space the European X-ray Multi Mirror (XMM) satellite. Its first test launch, however, on June 4, 1996 resulted in failure, with the rocket exploding 40 seconds into the flight.
A study of the accident by an inquiry board[4] found that the failure was caused by a software problem in the Inertial Reference System that was guiding the rocket. In particular, the computer program, which was written in the Ada programming language and was inherited from the previous launch vehicle Ariane 4, required at some point in the calculation a conversion of a 64-bit floating point number to a 16-bit integer. The initial trajectory of the Ariane 5 launch vehicle, however, is significantly different than that of Ariane 4 and the 16 bits are not enough to store some of the required information. The result was an error in the calculation, which the inertial system misinterpreted and caused the rocket to veer off its flight path and explode. The cost of the failed launch was upwards of 100 million dollars!

2.3 Rounding Errors

Relative Error

Any real number that does not terminate even when all the digits of the significand are used is, by construction, stored and manipulated by the computer only approximately. For example, the number $\pi = 3.1415927...$ can be stored approximately with 4 significant digits as $\bar{\pi} = 3.142$. We define the **relative error** with which a real number x is represented by an approximate floating point number $\bar{x}$ as the difference between the number and its approximate form, divided by the real number, i.e.,

$$(\text{relative error}) \equiv \frac{x - \bar{x}}{x} \; . \tag{2.4}$$

In the previous example, the relative error we introduce when we round the value of π to four significant digits is equal to $(\pi - 3.142)/\pi \simeq 0.00013$.

Machine Accuracy

The maximum possible relative error that may be introduced when a real number is represented by an approximate floating point number is called **machine accuracy**. Because we often use the symbol ϵ to denote the machine accuracy, we also refer to it as the machine epsilon. It can be shown that, if we use a numeral system of base b and keep p significant digits, the machine accuracy is equal to (see

reference [2] for the derivation)

$$\epsilon = \left(\frac{b}{2}\right) b^{-p} \ . \tag{2.5}$$

A single precision number, which requires 23 significant digits in the binary system (see Figure 2.1), corresponds to a machine accuracy of $\epsilon_{\rm single} = 2^{-23} \simeq 10^{-7}$. A double precision number, which requires 52 significant digits in the binary system, corresponds to a machine accuracy of $\epsilon_{\rm double} = 2^{-52} \simeq 2 \times 10^{-16}$.

2.3.1 Rounding errors in algebraic operations

A rounding error of 1 part in 10 million, as in the case of single precision numbers, might appear to be adequate for most practical problems in the physical world. However, arithmetic operations, such as additions and subtractions tend to amplify very quickly the rounding errors.

Consider for example the sum of three numbers $3.45 + 4.87 + (-5.16)$. Each of the numbers is given with 3 significant digits and is, therefore, known to an accuracy of ± 0.005. The sum, however, is equal to $(3.45 \pm 0.005) + (4.87 \pm 0.005) + (-5.16 \pm 0.005) = 3.16 \pm 0.015$ and hence this addition of only three numbers has already led to the loss of one significant digit. Numerical calculations often involve tens of millions of arithmetic operations, at the end of which the final result may be accurate to only a couple of digits or may not even be correct at all! This is the **fundamental limitation of all numerical computations**: unless special precautions are taken, the accuracy of a numerical calculation decreases rapidly with the number of arithmetic operations performed.

In the following examples, we will explore several situations that lead to loss of numerical accuracy in calculations and develop strategies to reduce or completely overcome them. We will start with the amplification of rounding errors encountered during the computation of sums.

Example: Counting Loops

Our first aim is to write an algorithm that computes the values of the function $f(x) = x^2$, when $0 \le x \le 1$ in steps of $\Delta x = 0.1$. First, we naively attempt the following few lines of code

```
#include <stdio.h>

int main(void)
{
   float x;

   for(x=0.0;x<=1.0;x+=0.1)
     printf("x=%f f(x)=%f\n",x,x*x);
   return 0;
}
```

The output of this little algorithm, however, is

```
x=0.000000 f(x)=0.000000
x=0.100000 f(x)=0.010000
x=0.200000 f(x)=0.040000
```

```
x=0.300000 f(x)=0.090000
x=0.400000 f(x)=0.160000
x=0.500000 f(x)=0.250000
x=0.600000 f(x)=0.360000
x=0.700000 f(x)=0.490000
x=0.800000 f(x)=0.640000
x=0.900000 f(x)=0.810000
```

Clearly something has gone wrong. The loop terminated when `x=0.9` and not when `x=1.0` as required by the condition in the `for` statement. Changing the `printf` statement to

```
printf("x=%10.8f f(x)=%10.8f\n",x,x*x);
```

reveals the problem. The output now is

```
x=0.00000000 f(x)=0.00000000
x=0.10000000 f(x)=0.01000000
x=0.20000000 f(x)=0.04000000
x=0.30000001 f(x)=0.09000001
x=0.40000001 f(x)=0.16000000
x=0.50000000 f(x)=0.25000000
x=0.60000002 f(x)=0.36000003
x=0.70000005 f(x)=0.49000007
x=0.80000007 f(x)=0.64000011
x=0.90000010 f(x)=0.81000017
```

Rounding errors have accumulated in the variable `x` such that the next value would have been `1.00000012`. This does not satisfy the condition in the `for` statement and hence the loop terminates. But why is there a rounding error when, for example, 0.1 is added to 0.2? The reason lies in the fact that the number 0.1 cannot be represented exactly in the binary system, which is what the digital computer uses to perform the addition. Indeed, as we have seen earlier, the binary representation of the decimal number 0.1 is $1.10011001100... \times 2^{-4}$, which does not terminate after the first 23 digits. As a result, adding the decimal number 0.1 to any number is an approximate calculation in floating point arithmetic.

Programming Tip

An obvious way to reduce such inadvertent rounding errors is the use of increments that are inverse powers of 2. For example, changing the loop in the above example to

```
for(x=0.0;x<=1.0;x+=1.0/16.0)
  printf("x=%f f(x)=%f\n",x,x*x);
```

will resolve the problem and the loop will terminate only after printing the result for `x=1.0`. In this case, the increment of the loop in the binary system is exactly equal to $(0.0001)_2$ and hence there is no rounding error involved.

Relying on using round numbers in the binary system, however, cannot be the general practice as nature does not follow in general the binary system! A more general solution to the problem involves using a variable of type `int` to count the number of steps taken. We can rewrite the above example to make use of this technique as

```
#include <stdio.h>

int main(void)
{
   int counter;
   float x;
```

```
   for(counter=0;counter<=10;counter++)
     {
       x=counter*0.1;
       printf("x=%f f(x)=%f\n",x,x*x);
     }
   return 0;
}
```

This loop will always terminate at the correct value for the variable `x` because there is no rounding error in incrementing a variable of type `int`.

In the previous example, the rounding error did not affect significantly the accuracy of any calculation. It introduced, however, a logical error that caused the algorithm to terminate before it was expected to. We will study now an example, in which the accuracy of the final result is significantly degraded when a large number of operations is performed.

Example: Computation of large sums

Consider the situation in which we have obtained a very large number of measurements of a particular quantity x and we would like to compute their average

$$\langle x\rangle = \frac{1}{N}\sum_{i=1}^{N} x_i\ . \qquad (2.6)$$

In this expression, x_i is the value of the $i-$th measurement and N is the total number of terms in the sum. In order to concentrate only on the propagation of rounding errors, we will perform the calculation for the trivial case in which each term is exactly equal to 0.1. In this case, $\langle x\rangle = 0.1$, independent of the number of terms added. However, as we have seen earlier, the binary representation of the decimal number $(0.1)_{10}$ requires an infinite number of binary digits and, therefore, each calculation introduces a rounding error. We will be using variables of type `float`, for which the machine accuracy is $\simeq 10^{-7}$.

We will perform the calculation using the following lines of code that evaluate the direct sum (2.6):

```
#include <stdio.h>
#define NTERM 10              // number of terms
#define VALUE 0.1             // value of each term

int main(void)
{
   int index;
   float sum=0.0,error;
                              // for all terms
   for(index=1;index<=NTERM;index++)
      sum+=VALUE;             // add each values
   sum/=NTERM;                 // divide by total number of terms
   error=(sum-VALUE)/VALUE;  // relative error
   printf("Sum=%g  Relative error=%g\n",sum,error);
   return 0;
}
```

Rounding errors may have deadly consequences

On February 25, 1991, during Operation Desert Storm, a Scud missile fired by the Iraqi army hit a U. S. army barracks in the Dhahran Air Base in Saudi Arabia, killing 28 soldiers. The air base was protected by a Patriot Air Defense System, which nevertheless failed to track and intercept the incoming Scud missile. An investigation of the incident by the U. S. army revealed that accumulation of rounding errors in the Patriot tracking software was responsible for the failure of the interception[5].
Each battery of the Patriot Air Defense system consists of a ground based radar and eight missile launchers. The radar detects airborne objects and tracks their motion. In order to distinguish between different types of objects and, in particular, in order to identify ballistic missiles, a weapons control computer calculates the expected trajectory of the missile. If the airborne object detected by the radar follows the expected trajectory, a Patriot missile is fired to intercept it.
The weapons control computer uses an internal clock to keep track of time and perform the calculations of the trajectories of the detected objects. The time is stored in 24-bit variables (registers) in increments of 0.1 seconds. In the binary system, however, $(0.1)_{10} = (0.0001100011...)_2$ and hence any calculation involving this time increment needs to be rounded. On February 25, 1991, after approximately 300 hours of continuous operation, the control computer had accumulated enough rounding error that it did not predict accurately the trajectory of the Scud missile. As a result, the missile was not identified and a Patriot was not fired to intercept it.

In this code, `NTERM` is the total number of terms and `VALUE` is the value of each one. At the end of the calculation, the algorithm prints the average as well as the relative error with respect to its true value.

Figure 2.2 shows the absolute value of the relative error in the calculation of the average, as a function of the number of terms in the sum. Because of rounding errors, the relative error in the average value is sometimes negative, sometimes positive, and when the various rounding errors cancel out, it can even be very close to zero. In the figure, the relative error appears to a have a quasi-periodic dependence on the number of terms only because all terms are equal to each other. In a more general situation, in which each term of the sum is different from the others, this would not be the case.

Overall, the absolute value of the relative error increases rapidly as the number of terms in the sum increases. When the number of terms grows to $\geq 10^8$, the sum takes values as large as $10^8 \times 0.1 = 10^7$, before the final division with the number

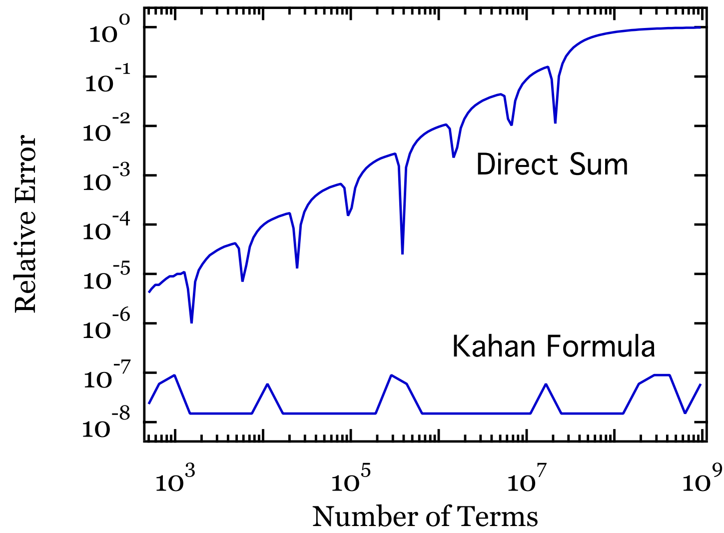


Figure 2.2: The absolute value of the relative error in the calculation of the sum (2.6) as a function of the number of terms included. The value of each term is set equal to 0.1. The curve on the top shows the amplification of rounding errors for the naive algorithm that computes the direct sum. The curve at the bottom shows the significant improvement introduced by the Kahan summation formula.

of terms N. At this point, the addition of more terms does not increase the value of the sum, since the ratio of each term to the sum is smaller than the machine accuracy. Indeed, $10^7 + 0.1 \simeq 10^7$, when the machine accuracy is 10^{-7}, as in the case of single precision numbers. Beyond this point, the sum does not increase proportionally to the number of terms, the value of the calculated average sharply decreases, and the relative error becomes equal to unity. The simple calculation of an average produced a result with 100% error, even though each term of the sum was larger than the machine accuracy by many orders of magnitude.

In order to improve the accuracy with which we evaluate a large sum, we can, in principle, perform all the computations with variables of type `double`, for which the machine accuracy is $\simeq 2 \times 10^{-16}$. We will study here, however, another technique often referred to as the Kahan summation formula[6], which will expose in a lucid way the problems of the direct summation algorithm we studied in the previous example.

Kahan Summation Formula

In general, the direct sum

$$S_{\rm N} = \sum_{i=1}^{N} x_i \tag{2.7}$$

can be cast in the form of the recursive formula

$$\begin{aligned} S_1 &= x_1 \\ S_{i+1} &= S_i + x_{i+1} , \end{aligned} \tag{2.8}$$

where S_i is the sum up to the i-th term. In the Kahan algorithm, the sum is corrected at each step according to

$$\begin{aligned} x'_{i+1} &= x_{i+1} - c_i \\ S_{i+1} &= S_i + x'_{i+1} , \end{aligned} \tag{2.9}$$

where c_i is a correction factor evaluated at the previous step with the recursive

relation

$$\begin{aligned} c_1 &= 0 \\ c_i &= [(S_{i-1} + x_i') - S_{i-1}] - x_i' \;. \end{aligned} \tag{2.10}$$

Mathematically speaking, all the terms at the right-hand-side of the last equation cancel out identically, leaving $c_i = 0$ at all times. This is not true, however, in floating point arithmetic, if we enforce the result of the operation in each parenthesis of bracket to be rounded before the next operation is performed.

In floating point arithmetic, the term $(S_{i-1}+x_i')$ is first evaluated and the result is rounded. Subtracting from this the original value of S_{i-1} leaves as a result only the part of the term x_i' that contributed to the rounded sum. Subtracting finally from this the original value of x_i' returns the part of the term x_i' that was rejected during the rounding process. This is the correction factor c_i that will be applied at the next step of the recursive process in an attempt to add back to the sum the part of the last term that was discarded in the rounding process.

We can rewrite now the algorithm of the previous example in order to evaluate the sum using the Kahan summation formula as

```
#include <stdio.h>
/* evaluate a large sum with the Kahan
           summation formula  */
#define NTERM 10             // number of terms
#define VALUE 0.1            // value of each term

int main(void)
{
   int index;
   float xprime, c=0.0, sum_interm;
   float sum,error;

   sum=VALUE;                // first term
                             // for all other terms
   for(index=2;index<=NTERM;index++)
      {
         xprime=VALUE-c;            // correct each term
         sum_interm=sum+xprime;     // temporary storage
         c=(sum_interm-sum)-xprime; // new correction
         sum=sum_interm;
      }
   sum/=NTERM;               // divide by total number of terms
   error=(sum-VALUE)/VALUE;  // relative error
   printf("Sum=%g  Relative error=%g\n",sum,error);
   return 0;
}
```

Figure 2.2 demonstrates the improvement in the calculation of a large sum that is offered by the Kahan summation formula. Indeed, the rounding errors do not get amplified and, even after the addition of hundreds of millions of terms, the relative error in the average is at most equal to the machine accuracy. In a later chapter, the Kahan formula will prove valuable in dealing with rounding errors when we will study the numerical evaluation of definite integrals.

Floating Point Addition is not Associative

The success of the Kahan summation formula relies on the fact that floating point addition is not associative, i.e.,

$$(a+b)+c \neq a+(b+c) \tag{2.11}$$

If this were not the case, then the correction term in equation (2.10) would be equal to

$$c_i = [(S_{i-1}+x'_i)-S_{i-1}]-x'_i = (S_{i-1}-S_{i-1})+(x'_i-x'_i) = 0 \tag{2.12}$$

and there would be no improvement in the accumulation of errors compared to the direct summation. The Kahan summation formula is indeed exploiting the non-associative character of floating point addition to correct for the rounding errors.

There are several situations, however, in which changing the order of evaluation of algebraic operations leads to results that are qualitatively and quantitatively different. Consider, for example, the simple sum

$$I = e^\tau - (e^\tau - 1)\ , \tag{2.13}$$

which is, of course, equal to unity, independent of the value of τ. If we evaluate this sum with single-precision floating numbers, we obtain $I = 1$ only when $\tau <= 44$. For larger values of the parameter τ, the term e^τ is much larger than one and hence $(e^\tau - 1) \simeq e^\tau$ because of rounding. As a result, $I \simeq e^\tau - e^\tau = 0$, which has a relative error of 100%! This is, in fact, a general problem of computations that involve the addition or subtraction of numbers that have very different magnitudes. In the following example (taken from reference [7]), we will study this problem in calculating the roots of a simple quadratic equation.

Example: Roots of the quadratic equation

In Chapter 1 (page 1-15), we developed a simple algorithm that calculates the roots of the quadratic equation.

$$ax^2 + b^x + c = 0\ . \tag{2.14}$$

When $a \neq 0$, and the discriminant $\Delta \equiv b^2 - 4ac \geq 0$, this equation has the two roots

$$x_{1,2} = \frac{-b \pm \sqrt{b^2-4ac}}{2a}\ . \tag{2.15}$$

In order to explore how rounding errors affect the computation of these roots, we will consider the case for which the parameter b is positive and $b^2 \gg 4ac$, so that the discriminant is always positive. Figure 2.3 shows the root with the positive sign as a function of the parameter b, when $a = -1$ and $c = 1$. The blue line corresponds to the value of the root calculated using single-precision floating point arithmetic, whereas the red line corresponds to the true value of the root. When the values of the three parameters, a, b, and c are all comparable to each other, the numerical computation of the root is well behaving. However, when the parameter β is significantly larger than the others, the rounding process leads to unacceptable errors.

We can understand the cause of the large rounding errors when $b^2 \gg 4ac$ by first approximating the square root of the discriminant as

$$\sqrt{\Delta} = (b^2-4ac)^{1/2} = b\left(1-\frac{4ac}{b^2}\right)^{1/2} \simeq b - \frac{2ac}{b} + \dots\ . \tag{2.16}$$

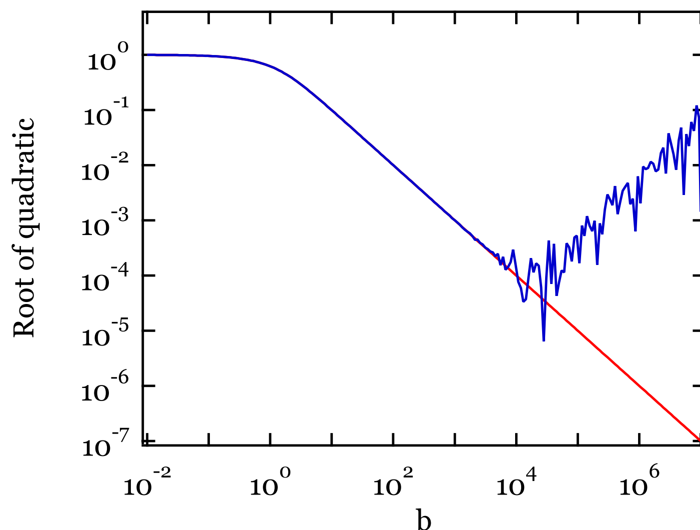


Figure 2.3: One of the roots of the quadratic equation $-x^2 + bx + c = 0$ plotted as a function of the parameter b. The red line shows the true value of the root, whereas the blue line shows the value calculated with single-precision floating point arithmetic using equation (2.15). For large values of the parameter b, the floating point computation introduces very large errors.

In the last step, we used the approximation $(1+\epsilon)^n \simeq 1 + n\epsilon + ...$, which is valid when $\epsilon \ll 1$. The two roots of the quadratic equation then become

$$x_1 = \frac{-b+\sqrt{b^2+4ac}}{2a} \simeq \frac{-b+b-\frac{2ac}{b}}{2a} = \frac{c}{b} \tag{2.17}$$

and

$$x_2 = \frac{-b-\sqrt{b^2+4ac}}{2a} \simeq \frac{-b-b-\frac{2ac}{b}}{2a} = -\frac{b^2+2ac}{2ab} \simeq -\frac{-b}{2a}\ . \tag{2.18}$$

Equation (2.17) reveals the problem. The computation of the numerator $-b+\sqrt{\Delta}$ involves the subtraction of two very large numbers, b and $\sqrt{\Delta}$, that have approximately equal magnitudes. The result of this subtraction is the very small number $4ac/b$. Therefore, even a small relative error in either of the two numbers in the subtraction leads to a very large relative error in the result. This large amplification of relative errors during the subtraction of two large but comparable numbers is said to be caused by **catastrophic cancellation**.

Catastrophic Cancellation

The only certain way of avoiding the effects of catastrophic cancellation is by manipulating mathematically the troubling expression in order to remove the subtraction. For example, in the case of the quadratic equation that we studied above, we can multiply the numerator and denominator of the root with the positive sign by $-b-\sqrt{\Delta}$ to obtain

Programming Tip

$$x_2 = -\frac{2c}{b+\sqrt{b^2-4ac}}\ . \tag{2.19}$$

This expression is mathematically equivalent to equation (2.15) for the root with the positive sign. However, when $b^2 \gg 4ac$, it becomes

$$x_2 = -\frac{2c}{b+b-\frac{2ac}{b}} \simeq \frac{c}{b}\ . \tag{2.20}$$

and does not involve the subtraction of large but comparable numbers. This mathematical manipulation resolved the effects of catastrophic cancellation.

2.4 Special Situations

In mathematics, not every operation between two real numbers results in a real number. For example, the square root of a negative number, e.g., $\sqrt{-i}$, is imaginary. Moreover, some algebraic operations, such as the division of a real number by zero, lead to infinities. Early versions of most computer languages would cause a program to halt when either of the two situations occurred during a numerical computation. However, the IEEE 754 standard requires that a computer language does not halt the calculation, but rather that it generates a special result. These situations are called **floating point exceptions**.

Floating Point Infinity

Infinity, in floating point arithmetic, is any real number that is too large to be stored in a variable of a particular type. Consider, for example, the following simple program

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

int main(void)
{
   float exponent,result;
   for (exponent=1.0;exponent<=130.0;exponent++)
     {
       result=pow(2.0,exponent);
       printf("exponent=%f result=%g\n",a,b);
     }
   return 0;
}
```

which prints on the screen successively increasing powers of two. When the variable `exponent` attains very large values, the result of the operation `pow(2,exponent)` becomes too large to be stored in a variable of type `float`. The calculation, however, is not stopped, but rather the last four lines of the output on the screen are

```
exponent=127.000000 result=1.70141e+38
exponent=128.000000 result=inf
exponent=129.000000 result=inf
exponent=130.000000 result=inf
```

Note that, according to the IEEE 754 standard, infinity may have a positive or negative sign. If we change for example the line of the above program that performs the calculation to

```
result=-pow(2.0,exponent);
```

the last four lines of the output will be

```
exponent=127.000000 result=-1.70141e+38
exponent=128.000000 result=-inf
exponent=129.000000 result=-inf
exponent=130.000000 result=-inf
```

There are two main reasons for which the IEEE 754 standard requires that a computer program is not stopped by a floating point exception but is allowed to continue. First, in several critical situations, a well written computer program can identify the reason that led to a floating point exception and correct it appropriately. Consider for example the failure of the Ariane 5 rocket discussed in the box of page 2-7. In such a situation, halting the execution of the program that controls the guidance of a rocket is catastrophic, whereas identifying and correcting the problem can save a very expensive launch from failure.

Second, there are several well defined arithmetic operations with infinities that result in real numbers. For example, the function

$$f(x) = \frac{1}{1+x} \tag{2.21}$$

is well defined and, in fact, is equal to one, when $x \to \infty$. Halting the execution of a computer program in such a case would be counterproductive. In order to investigate whether this generates a floating point exception or not, we can change the line of the above program that performs the calculation to

```
result=1.0/(1.0+pow(2.0,exponent));
```

The program will generate the correct result, even though the intermediate step `pow(2.0,exponent)` has resulted in an infinity for large values of the variable `exponent`.

NAN

We need, here, to be very careful when we allow for infinities to take place in numerical computations, since a computer language can not keep track of how fast an expression reaches infinity. For example, the function

$$g(x) = \frac{x+1}{x^2+x} \tag{2.22}$$

is well behaving in mathematics when $x \to \infty$ since

$$\lim_{x\to\infty} g(x) = \lim_{x\to\infty} \frac{1}{x+1} = 1 \; , \tag{2.23}$$

even though both the numerator and the denominator become very large. This is not true, however, in floating point arithmetic. Changing the line in the above program that performs the calculation to

```
result=pow(2.0,exponent);
result=result/(result*result+result);
```

returns another special code, `nan`. This special code stands for **n**ot-**a**-**n**umber and occurs when the result of an algebraic computation is not well defined or is not a real number Operations that return `nan` involve calculations of the form $\infty - \infty$, $0 \times \infty$, $0/0$ and ∞/∞, all of which are undefined in mathematics. Moreover, operations such as taking the square root or the logarithm of a negative number also result in `nan`. Contrary to the case of infinities, no calculation that involves the special code `nan` can have a real number as a result. If we anticipate that, under some special circumstances, such a floating point exception may occur, we need to protect our program from returning a `nan` via the use of branching statements such as `if` or `do while`.

Further Reading

1. *The Art of Computer Programming. Volume II: Semi-Numerical Algorithms*, by D. A. Knuth, (Addison-Wesley), 3rd ed., 1977

2. *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, by D. Goldberg, in the March 1991 of *Computing Surveys*, available online at http://docs.sun.com/source/806-3568/ncg_goldberg.html

3. *IEEE 754: Standard for Binary Floating Arithmetic*, can be found at http://grouper.ieee.org/groups/754/

4. *Ariane 5: Flight 501 Failure*, Report by the Inquiry Board chaired by Prof. J. L. Lions, July 19, 1996

5. *Roundoff Error and the Patriot Missile* by R. Skeele in SIAM News, 25, 4 (1992)

6. *Further Remarks on Reducing Truncation Errors* by W. Kahan, Communications of the ACM 8, 1, 40 (1965)

7. *Pitfalls in Computation, or Why a Math Book Isn't Enough* by G. E. Forsythe, The American Mathematical Monthly, 77, 931 (1970)