Chapter 2

Floating-point Computing

"*Floating-point numbers are like piles of sand; every time you move them around, you lose a little sand and pick up a little dirt.*" – Dr. Brian Kernighan (Princeton professor and contributor of UNIX) and Dr. P. J. Plauger (entrepreneur and author of many programming books).

Computers use a fixed number of bits to represent a piece of data, which could be a number, a character, etc. A n-bit storage location can represent up to 2^n distinct entities. For example, a 32-bit computer can hold unsigned integers ranging from 0 to 4,294,967,295 ($= 2^{32} - 1$). Integers provide an exact representation for numeric values. However, they suffer from two major drawbacks: the inability to represent fractional values and a limited dynamic range. Floating-point arithmetic solves these two problems at the expense of accuracy and speed. Some of the greatest achievements of the 20th century would not have been possible without the floating-point capabilities of digital computers. Nevertheless, this subject is not well understood by software developers and is a regular source of confusion. In a February, 1998 keynote address entitled *Extensions to Java for Numerical Computing*, Dr. James Gosling, the creator of Java programming language, asserted "*95% of folks out there are completely clueless about floating-point.*" His assertion may sound far-fetched to many but has good ground in consideration of the following examples:

- In 1982 the Vancouver Stock Exchange instituted a new index much like the Dow-Jones Index. It began with a nominal value of 1,000.000 and was recalculated based on the selling price of all listed stocks after each recorded transaction. This computation was done using four decimal places and truncating (not rounding) the result to three. Stock transactions happen 2,800 times a day. After 22 months of accumulated roundoff error, the Vancouver Stock Exchange index was undervalued by almost 50% even though the exchange was setting records for volume and value. When this computation problem was found and corrected by rounding off (not truncating) the fourth decimal place, the index sat at 1098.892.

- On February 25, 1991 an American Patriot missile failed to track and destroy an Iraqi Scud missile. Instead it hit an Army barrack, killing 26 people. The cause was later determined to be an inaccurate calculation caused by measuring time in

a tenth of a second that cannot be represented exactly since a 24-bit floating-point was used.

- Ariane 5's first test flight on 4 June 1996 failed, with the rocket self-destructing 37 seconds after launch because of a malfunction in the control software. The malfunction was traced to unanticipated overflow in converting 64-bit floating-point number to a 16-bit signed integer.

- A few years ago, the company where I work started to use the commercialized static code analysis tool Coverity® to automatically test source code for software defects that could lead to product crashes, unexpected behavior, security breaches, or catastrophic failures. One of the top five defect criterions was the direct comparison of floating-point values for equality, which, as a rule of thumb, should rarely be used in floating-point computing.

- Geometric models created at different computer-aided design (CAD) systems may use different units for measurement. Commonly-used units are the millimeter, meter, inch, foot, etc. When a model generated in one system is imported to another system, it is important to use a correct scaling factor to scale the model geometry to obtain consistent measurements. For example, to convert a drawing created in millimeters to meters, the scaling factor of 1/1,000 needs to be used. AutoCAD® (developed by Autodesk) is probably the only popular CAD system that allows users to create the so-called *unitless* drawings and models. It is, however, emphasized in a 2007 *Autodesk University*'s article that "this (unitless) selection can only make sense if you never ever switch between metric and imperial, or never exchange files with other companies." If a unitless model has to be imported to a different system, it is important to derive a scaling factor based on the allowed workspace and properly scale the model to maintain model integrity. Unfortunately, this is not always the case. As a manager and developer at Math and Geometric Modeling group, I have seen numerous cases in which unitless models were imported without scaling. Therefore, a model may sit so far away that the distance between the model and the origin of coordinate system is equivalent to the distance from the Moon to the Earth (roughly 4×10^8 meters). When geometric operations fail to create consistent results with respect to the tolerance of 10^{-6} meter, people blame the "unstable" math and geometric algorithms. Apparently, those developers who imported the foreign models are unaware of floating-point representation in a digital computer and hence fail to understand that a `double` data type in a 32-bit machine has at most 16 significant digits. Eight or more significant digits are required to represent a model sitting on the Moon. Additional six digits are required to describe the measurement with accuracy in micrometer (i.e., 10^{-6} meter). Therefore, 14 or more significant digits will be consumed, leaving no room for round-off error that occurs in numerical computation and approximation.

Floating-point arithmetic is a very important subject, and a rudimentary understanding of it is a pre-requisite for programmers, operating system designers, programming language and compiler writers. For this reason, floating-point representation, arithmetic, and limitations will be reviewed in this chapter prior to any discussion of numerical method.

Digital computers cannot represent all real numbers exactly, so we are constantly facing challenges when designing computer algorithms for real number computations. Such challenges are further exacerbated since many important scientific algorithms make additional approximations when exact solutions are not practical. It is not uncommon that a targeted computation result can be achieved in several ways, all of which may be equivalent in theory, but in practice when performed on digital computers yield different results. Some calculations might dampen out approximation errors (also known as *numerical noises*); others might magnify such errors. Calculations that can be proven not to magnify approximation errors are referred to as *numerically stable*. In this chapter, we will also discuss general rules in reducing numerical errors and how to determine if an algorithms is robust based on some examples.

2.1 Floating-point representation

A modern digital computer represents data using the binary numeral system. Text, numbers, pictures, audio, and nearly any other form of information can be converted into a string of *bits*, or *binary digits*, each of which has a value of 1 or 0. The most common unit of storage is the *byte*, which is equal to 8 bits.

Computers use a fixed number of bits to represent a piece of data, which could be a character, an integer, a double, etc. An n-bit storage location can represent up to 2^n distinct entities. For example, a 2-bit memory location can hold one of these four binary patterns:

$$00, \quad 01, \quad 10, \quad 11.$$

Hence, it can represent at most 4 distinct entities (e.g., integer numbers 0 to 3). Modern personal computers have either 32-bit or 64-bit architectures that can hold unsigned integers ranging

32-bit	0 to 4,294,967,295 ($= 2^{32} - 1$).
64-bit	0 to 18,446,744,073,709,551,615 ($= 2^{64} - 1$).

These ranges are "huge" for counting but very limited for real world computing. For example, the factorial of 21 is

$$21! = 51,090,942,171,709,440,000$$

which means that a 64-bit computer would not be able to compute factorials of 21 or higher if it has only integer representation. Lack of dynamic range is not the only limitation of integer representation. Another significant drawback of integer presentation is the inability to represent fractional values.

The term floating-point refers to the fact that its decimal point can "*float*" anywhere relative to the significant digits of the number with the help of exponents. For example, we can let the decimal "float" by writing 3141.59 in *scientific notation* as

$$314.159 \times 10, \quad 31.4159 \times 10^2, \quad 3.14159 \times 10^3$$

with the last one being also called the *normalized scientific notation*. In scientific notation all real numbers are written in the form of

$$m \times 10^n$$

where the exponent n is an integer, and the coefficient m is any real number called the *mantissa* or *significand*. When $1 \leq |m| < 10$, the notation is then the normalized scientific notation.

The significand (or mantissa) is part of a number in scientific notation, consisting of its *significant digits*. The significant digits of a number are those digits that carry meaning contributing to its accuracy. Every science and engineering student knows that no measurement of a physical quantity can be entirely accurate. It is important to know, therefore, just how much the measured value is likely to deviate from the unknown, true value of the quantity. For example, if I say I weigh 135 lbs, what I am really saying is:

$$134.5\,\text{lbs} < \text{my weight} < 135.5\,\text{lbs}$$

This is because I have confidence that my scale is accurate to within a half pound, and that I read the number correctly off the dial. But I don't say that I weigh 135.000 lbs, since that would imply extraordinary precision of my scale:

$$134.9995\,\text{lbs} < \text{my weight} < 135.0005\,\text{lbs}$$

By saying I weigh 135 lbs, I imply that three digits in the number are known to be correct. Accordingly, these three digits are significant.

Non-zero digits are always significant. With zeroes, the situation is more complicated:

1. Zeroes placed between other digits are always significant; 5004 has four significant digits and 601 has three.

2. Zeroes placed before other digits are not significant. They are simply placeholders. For example, the diameter of my watch without crown is 28 millimeters. This measurement has two significant digits. If the diameter is written in meter, it is then 0.028 meter. Zeros are simply placeholders. Therefore, 0.028 has only two significant numbers.

3. Zeroes placed after other digits but behind a *decimal point* are significant. If there is no decimal point, they are usually considered as placeholders and hence are not counted. For example, when you are told that the population in Huntsville, Alabama is approximately 181000, it often means that the figure is rounded to the nearest thousand. Therefore, 181000 has only three significant digits. However, 3.700 has four significant digits since zeroes are placed after a decimal point to indicate a measurement precise to three decimal places.

To avoid confusion and make counting easy, always write the number in the normalized scientific notation. In this case, all digits are significant.

Floating-point notation in computers is analogous to scientific notation for decimal numbers. Suppose

$$x = \pm m \times 2^E$$

where $1 \le m = (b_0 b_1 b_2 \cdots)_2 < 2$ and $b_0, b_1, \cdots$ are bits. To store a number in floating-point representation, a computer word is divided into 3 fields, representing the sign (0 for positive and 1 for negative), the exponent E, and the significand (or mantissa) m respectively. A 32-bit word is used for the *single-precision* floating-point format (known as `float` in C/C++) and is divided into fields as follows: 1 bit for the sign, 8 bits for the exponent and 23 bits for the significand (23 bits is equivalent to $\log_{10} 2^{23} \approx 7$ significant decimal digits). Since the exponent field is 8 bits, it can be used to represent signed exponents between -126 and 127 (approximately between -38 and 38 in base 10 since $2^{-126} \approx 1.1755 \times 10^{-38}$ and $2^{127} \approx 1.7014 \times 10^{+38}$). The significand field can store the first 23 bits of the binary representation of m, namely

$$b_0 b_1 b_2 \cdots b_{22}.$$

To store a base-10 number in computer, we first convert the integer part to its binary equivalent. This may be done by repeatedly dividing the integer by two until the quotient becomes zero. Depending on the dividend is odd or even, we write respectively the remainder 1 or 0 on the right of each division process. The binary equivalent is then obtained by reading the sequence of remainders upwards to the top. For example, the processes to convert $(71)_{10}$ are

$$\begin{array}{rl} 2\underline{|71} & 1 \\ 2\underline{|35} & 1 \\ 2\underline{|17} & 1 \\ 2\underline{|8} & 0 \\ 2\underline{|4} & 0 \\ 2\underline{|2} & 0 \\ 2\underline{|1} & 1 \end{array}$$

We thus have $(71)_{10} = (1000111)_2$ or, in the normalized form, $(1.000111)_2 \times 2^6$. Accordingly, it is stored in 32-bit computer as

0	$E = 6$	1.00011100000000000000

Reversely,we can convert $(1000111)_2$ to the its decimal equivalent as follows:

$$1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = (71)_{10}.$$

Note that the exponent $E = 6$ should actually be stored in binary numbers as well. For now, we use decimal number 6 for simplicity.

Converting a decimal fraction to binary number is done by repeatedly multiplying the decimal fraction by 2. The whole number (either 1 or 0) is the binary number repeatedly appended to the right of the point. For example, the steps to convert 0.625 to the binary representation are

- $0.625 \times 2 = 1.25$, the first binary digit to the right of the point is a 1.
- Discard the whole number part of the previous result and multiply the fraction by 2: $0.25 \times 2 = 0.5$. So, the second binary digit to the right of the point is a 0.
- Multiply the fraction by 2 again: $0.5 \times 2 = 1.0$. So, the thrid binary digit to the right of the point is a 1.
- Terminate the process since the fraction part is now zero.

Therefore, we have $(0.625)_{10} = (.101)_2$ or, in the normalized form, $(.101)_2 = (1.01)_2 \times 2^{-1}$. Reversely, we can convert the binary number to the decimal fraction as follows:

$$0.625 = 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}.$$

Since computers have only a finite number of bits for storing a real number, any number that has an infinite number of digits such as $1/3$, $\sqrt{2}$ and π cannot be represented completely in computer. It may surprise some people that even a number with finite decimal digits cannot be represented precisely because of the way of encoding real numbers. For example, when 0.1 in base 10 is represented in a binary system, it is

$$(1.100110011\cdots)_2 \times 2^{-4},$$

which is stored in a 32-bit machine with truncation as:

0	$E = -4$	1.10011001100110011001100

Thus, there is loss of precision. In general, a fraction with finite decimal expansions can be represented as a rational number a/b. This rational number has finite binary expansions if and only if the denominator has 2 as the only prime factor, i.e., $b = 2^n$. It should be pointed out that a fraction number that has finite binary expansions may still have to be stored as an approximation because it may have more binary expansions than the available bits can hold.

We now discuss representation of exponents in computers. In IEEE 754 floating-point numbers, the exponent is *biased* in the engineering sense of the word – the value stored is offset from the actual value by the exponent bias. Biasing is done because exponents have to be signed values in order to be able to represent both tiny and huge values, but two's complement, the usual representation for signed values, would make comparison harder. To solve this problem, the exponent is biased before being stored, by adjusting its value to put it within an unsigned range suitable for comparison. For example, if the real exponent of a number is X, then it is represented as $(X + \text{bias})$. When interpreting the floating-point number, the bias is subtracted to retrieve the actual exponent. IEEE single-precision uses a bias of 127. Therefore, an exponent of

-4 is represented as $-4 + 127 = 123 = (01111011)_2$

0 is represented as $0 + 127 = 127 = (01111111)_2$

$+1$ is represented as $+1 + 127 = 128 = (10000000)_2$

$+6$ is represented as $+6 + 127 = 133 = (10000101)_2$

After understanding the representation of an exponent, we can convert 71 in base 10 to binary system to store in computers as

0	10000101	1.00011100000000000000

Similarly, 0.1 would be

0	01111011	1.10011001100110011001 10

For most programmers, it may not be so important to know the exact representation of floating-point numbers. It is, however, very important to remember that a computer has only a finite number of bits for storing the exponent and significand. Therefore,

- When the result of a calculation is too large to represent in a floating-point number system, we say that *overflow* has occurred. This happens when a number's exponent is larger than allowed.
- Just as overflow occurs when an exponent is too big, *underflow* occurs when an exponent is too small.
- Finite number of bits for storing the significand requires an infinite number of digits to be truncated. Such roundoff errors may be further exacerbated by floating-point arithmetic in the numerical process, which will be discussed in subsequent sections.

It should be pointed out that a *double-precision* floating-point format (known as `double` in C/C++) is more widely used than single-precision in numerical computations in order to retain maximum significant digits in spite of its performance and bandwidth cost. Computers with 32-bit storage locations use two memory locations to store a 64-bit double-precision number. The 64-bit is divided into three fields as follows: 1 bit for the sign, 11 bits for the exponent and 52 bits for the significand (52 bits is equivalent to $\log_{10} 2^{52} \approx 15 \sim 16$ significant decimal digits). Since the exponent bias for a double-precision floating-point format is 1023 and the exponent is stored as $X + \text{bias}$, the true exponent range is:

$$\begin{aligned} \mathrm{X_{min}} + \text{bias} = 1 \quad &\Rightarrow \quad \mathrm{X_{min}} = 1 - \text{bias} = -1022 \\ \mathrm{X_{max}} + \text{bias} = 2046 \quad &\Rightarrow \quad \mathrm{X_{max}} = 2046 - \text{bias} = 1023 \end{aligned}$$

Note that $2046 = 2^{11} - 2$ (one bit for the sign and one bit for zero). Converting the exponent range between -1022 and 1023 to the exponent for base 10 is approximately between -308 and 308 ($2^{-1022} \approx 2.225074 \times 10^{-308}$ and $2^{1023} \approx 1.0 \times 10^{+308}$).

2.2 Floating-point comparison

As a rule of thumb in programming, we should avoid direct comparison of two floating-point values for equality because computers have limited precision. It is widely regarded that direct comparisons of floating-point values are the sources of instability that can be triggered by arithmetic computations, change of compilers, or even change of compiling options. For this reason, any direct comparison of floating-point values is considered a defect by Coverity® and must be scrutinized carefully to determine if a defect truly

exists or if the comparison is being performed in an appropriate context. Some people may think that a solution to such anomalies is simply not to compare floating-point numbers for equality, but instead to consider them equal if they are within some error bound ε. This is hardly a cure-all, because it raises as many questions as it answers. What should the value of ε be? If the absolute values of $x < 0$ and $y > 0$ are within ε, should they really be considered equal, even though they have different signs? We will try to answer these questions in this section.

2.2.1 Machine precision

Because of the approximate representation of real numbers in computers, a direct comparison of equality as illustrated below is instable as it may perform `operation 1` now and `operation 2` in the future when an environment such as a compiler and operating system changes.

```
if (a == b)
{
   // operation 1
}
else
{
   // operation 2
}
```

A tolerance should be used while comparing two floating-point values. In floating-point arithmetic, the *machine epsilon* ε (also called *machine precision* or *machine tolerance*) is a very useful quantity in numerical error analysis. For any given format (e.g., `float`, `double`, or `long double`), the machine epsilon is the difference between 1 and the next larger number that can be stored in that format. For a single-precision floating format, there are 23 bits for the significand. Accordingly, the next larger binary number is

$$1.00000000000000000000001$$

which is $1 \times 2^0 + 1 \times 2^{-23} \approx 1 + 1.19 \times 10^{-7}$ in the decimal number system. Due to the restriction of the 23-bit significand, it is clear that $1 + 2^{-24}$ cannot be stored exactly. Therefore, the machine epsilon for a `float` is approximately 1.19×10^{-7}. Similarly, 52 bits are used for the significand of a double-precision floating-point format. Consequently, the machine epsilon for a `double` is $2^{-52} \approx 2.22 \times 10^{-16}$.

The machine epsilon can also be determined via a C program. For example, we can use the following C program to determine the machine precision for `double` data type:

```
double epsilon = 1.0;

while ((1.0 + epsilon) != 1.0)
{
   epsilon /= 2.0;
}
printf( "\nCalculated Machine epsilon: %.15e\n\n", 2 * epsilon );
```

The `while` loop breaks when adding `epsilon` to 1 no longer makes difference (i.e., the condition `1.0 + epsilon == 1.0` is true). Since ε should be the smallest positive floating-point number such that $1 + \varepsilon > 1$, we thus need to restore the previous value by printing the machine epsilon with `2 * epsilon`. Running the above program under Microsoft Visual C++ 2010 compiler, we would get

$$\varepsilon = 2.2204460492503131 \times 10^{-16},$$

which is 2^{-52}. Machine epsilons for `float` and `double` are available in C/C++ header file `float.h` as `FLT_EPSILON` and `DBL_EPSILON`.

2.2.2 Comparison with absolute error bound

The absolute difference $|a - b|$ is computed and compared to a fixed error bound ε. If the following condition is true,

$$|a - b| < \varepsilon$$

then a is considered equal to b. We must exercise with care when choosing the value for the error bound. It should be a value greater than the minimum representable difference for the range and type of float we are dealing with. For example, ε is recommended to be larger or equal to the machine epsilon when comparing two values whose types are `double`.

The absolute difference comparison might be used to identify numbers that are approximately the same. The advantages of comparisons with absolute are speed, simplicity and portability. A severe disadvantage is lack of consideration of the size of a number. For example, a 1 millimeter error in the length of a one-meter long household pipe is certainly more serious than a 1 millimeter error in a 1000-meter long oil transportation pipe. With absolute error bound, however, both cases will be treated equally. Furthermore, it pays no attention to differences in significant digits. Assume $a = 2.2204460 \times 10^{-16}$ and $b = 1.110223024625157 \times 10^{-16}$. By using the absolute error bound, we would consider them to be equal even though their *relative error*

$$\frac{|a - b|}{|a|}$$

is 0.5 (more in the next section).

2.2.3 Comparison with relative error bound

In many applications, we are interested in knowing the differences of significant digits rather than the absolute difference of two floating-point numbers. In this case, the following comparison makes more sense than the absolute error bound:

$$|a - b| < \frac{max(|a|, |b|)}{10^{15}}$$

If such condition is true, we say a and b are equal within 15 digits. In some publications, the following formula is used:

$$|a - b| < max(|a|, |b|) \times \varepsilon.$$

If neither $|a|$ nor $|b|$ is zero, we can write the above relation as:

$$\frac{|a-b|}{max(|a|,|b|)} < \varepsilon.$$

As is seen above, this is a relative error bound. In other words, the error is measured as the percentage of the size of the larger number. In general, the relative error makes more sense than the absolute error since it takes into consideration the size of the number. However, users need to pay special attentions to the following aspects:

1. When both a and b are very small, it may result in underflow without a safeguard in division by 10^{15} or multiplication by ε. The term floating-point underflow is a condition in a computer program where the result of a calculation is a number smaller than the computer can actually store in memory.

2. From a practical point of view, we would treat $a = 1.0 \times 10^{-30}$ as zero. However, it would not be considered equal to zero using the relative error bound since they do not have equal significant digits.

2.2.4 Geometric comparison

At Intergraph Corporation, we develop CAD/CAM software systems and work with geometric entities created by using parametric curves and surfaces all the time. In order to maintain model integrity, we must impose a set of tolerances to ensure proper interpretation of positions, such as determining whether two curves intersect, a position lies on a curve, or a position is inside or outside a volume. The following three tolerances are widely used to control modeling operations in CAD/CAM systems:

Distance tolerance If the Euclidean distance of two positions is less than this tolerance, we consider the two points coincident. By default, distance tolerance is set to 10^{-6} unit (e.g., meter).

Angular tolerance For 32-bit computer, a double-precision floating-point format uses 52-bit for the significand, which is equivalent to 15 or 16 significant decimal digits ($\log_{10} 2^{52} \approx 15.654$). We consider five of the least significant digits to represent numeric round-off errors that occur during calculations. Thus, there are roughly 10 digits that represent the dynamic range of numbers (smallest and largest numbers) within object space. Accordingly, the angular tolerance, which is the smallest of all three tolerances, is set to 10^{-10}.

Fit tolerance This tolerance is used when approximation is involved (e.g., fitting point cloud by spline curve via the least squares approximation method). By default, it is set to 10^{-3} unit.

When the distance and angular tolerances are defined, the longest line we can model would be

$$\frac{10^{-6}}{10^{-10}} = 10,000 \text{ units}$$

Reversely, the angular tolerance can be derived from the distance tolerance and the limit of model space. Since the longest line in the confined model space is 10^4 units,

the deviation incurred by rotating this line by a small angle (say, the angular tolerance Δ) is $10^4 \times \Delta$. If the rotated line is still considered to be the same as the original one with respect to the distance tolerance, the deviation has to be less than the distance tolerance, i.e.,

$$10^4 \times \Delta < \varepsilon.$$

Therefore, the angular tolerance is

$$\Delta < \frac{\varepsilon}{10^4} = 10^{-10},$$

which answers the second pre-interview question in the preface.

Sometime we need to derive a tolerance from the above three. The first pre-interview question in the preface asked for a derivation of a tolerance to measure the area of circle. It was assumed that the radius (R_1) and area (A_1) of circle 1 is known. We further assumed that the radius of the ith circle is $R_i = R_1 \pm \varepsilon$. Then, the area tolerance is given by

$$\Delta = |A_i - A_1| = \pi|(R_1 \pm \varepsilon)^2 - R_1^2| = \pi| \pm 2R_1\varepsilon + \varepsilon^2| = \pi|2R_1\varepsilon \pm \varepsilon^2|.$$

Since ε^2 is negligible, we have $\Delta = 2\pi R_1\varepsilon$. Calculus is the mathematical study of change. We may derive the area tolerance based on calculus. Since $A = \pi R^2$. By the rule of differentials,

$$dA = 2\pi R\, dR.$$

It is known that dA measures the change of area with respect to the small change of radius (i.e., dR). Since dR needs to be smaller than ε, we have $\Delta = 2\pi R\varepsilon$. Replacing R by the known R_1, we obtain $\Delta = 2\pi R_1\varepsilon$.

2.2.5 Round number to given decimal place

It is often desirable to round a number to a specified decimal place. Assuming we want to round 0.084849999 to the third decimal place to obtain 0.085, we would

- Multiply the number by 1000 to get 84.849999.
- Add 0.5 to round to the nearest integer. The result would be 85.349999.
- Truncate all decimals. The result is 85.0.
- Finally, divide the truncated number by 1000.

By generalizing the above steps, we have the following implementation:

```
double apiRoundValue(
    double  dValue,     // Input: the number to be rounded
    int     nDecimals)  // Input: the decimal digits to keep
{
   double   temp, dScale;

   // Scale the number and add 0.5 so the first decimal of
```

```
    // enlarged number is rounded to the nearest integer if
    // it's equal or larger than 0.5.
    dScale = pow(10.0, nDecimals);
    temp = dScale * dValue + 0.5;

    // Truncate the decimal numbers and scale it back
    temp = floor(temp);
    return temp / dScale;
}
```

As previously discussed, computers have only limited precision. Therefore, most numbers just cannot be presented exactly. We take the same number (0.084849999) as an example. Assume we want to round it to the fifth decimal place. The expected result is 0.08485. Instead, we will get

0.084849999999999995

when running Microsoft Visual C++ 2010.

2.3 Design numerically stable algorithms

Numerical stability is an important notion in numerical analysis. An algorithm is called numerically stable if an error, whatever its cause, does not grow to be much larger during the calculation. This happens if the problem is *well-conditioned* , meaning that the solution changes by only a small amount if the problem data are changed by a small amount. On the other hand, if a problem is *ill-conditioned* , then any small error in the data will grow to be a large error. The *condition number* is measured as

$$C_p = \frac{\text{relative change in solution}}{\text{relative change in input data}}$$

The condition number determines how sensitive a problem is to small perturbations of input values. If C_p is not much larger than 1 we call the problem well-conditioned; in the case of $C_p \gg 1$ we call the problem ill-conditioned. For a function $f(x)$, the condition number at x_0 is

$$C_p = \left| \frac{(f(x) - f(x_0))/f(x_0)}{(x - x_0)/x_0} \right| \approx \left| \frac{x_0 f'(x_0)}{f(x_0)} \right|$$

It is beyond the scope of this section to study well-conditioned and ill-conditioned systems in detail. For most developers, it is important to realize that floating-point mathematics (as implemented in all modern computer systems) is usually not exact but an approximation to the real number system. In practice we can improve numerical stability by paying special attention to the following floating-point arithmetic:

- Avoid dividing a large number by a very small one, as it is equivalent to multiplying a very large number to the numerical noise. If possible, try to divide numbers that have the same relative magnitudes.
- Avoid subtraction between two similar numbers. Subtracting two numbers that are almost equal results in great loss of significant digits. The numerical instability may

be understood by looking at the relative error of subtraction. Let $\bar{a} = a(1 + \Delta a)$ and $\bar{b} = b(1 + \Delta b)$ be two approximation numbers to a and b. The relative error caused by small perturbations Δa and Δb is

$$\frac{|\bar{x} - x|}{|x|} = \frac{|a\Delta a - b\Delta b|}{|a - b|},$$

where $\bar{x} = \bar{a} - \bar{b}$ and $x = a - b$. If $a \approx b$, small Δa and Δb can lead to very large relative error in x (known as the *cancellation error*).

- Avoid adding a very small number to a large number. Floating-point computation has limited precision. If a large float is added to a small float, the small float may be too negligible to change the value of the larger float. In performing a sequence of additions, the numbers should be added in the order of smallest in magnitude to largest in magnitude. This ensures that the cumulative sum of many small arguments is not negligible.

- Reduce arithmetic computations to minimize cumulative error. Each arithmetic operation adds numerical noise at least linearly if not exponentially (Please note some numerical noises may cancel each other out due to different sign). Therefore, reduction of arithmetic operations generally improves numerical stability.

- Select the most stable algorithm when several approaches are available.

- Transform geometric models to the origin before doing complex geometric computations. It is not uncommon for CAD system users to create their models far away from the origin of global coordinate system. In this case, any geometry will have large (x, y, z) components and hence leave fewer decimal digits for computations. Transforming them close to the origin will greatly improve the stability of numerical computation.

A few examples are given below to illustrate how numerical stability can be improved.

Example 1: In high school, we were all taught that solutions to a quadratic equation $ax^2 + bx + c = 0$ is

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, \quad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}.$$

The above solutions can be numerically unstable if $b^2 \gg 4|ac|$ because of subtraction between two similar numbers. In this case, it is better to utilize the identity $x_1 x_2 = c/a$ to calculate one root from another.

- If $b < 0$, calculate $x_1 = \dfrac{-b + \sqrt{b^2 - 4ac}}{2a}, \quad x_2 = \dfrac{c}{ax_1}$.

- If $b > 0$, calculate $x_2 = \dfrac{-b - \sqrt{b^2 - 4ac}}{2a}, \quad x_1 = \dfrac{c}{ax_2}$.

If you are not convinced, try the following code fragment

```
float x2, x1_bad, x1_good, delta;
delta = (float)sqrt(b * b - 4.0 * a * c);
x2 = (-b - delta)/(2.0 * a);
x1_bad = (-b + delta)/(2.0 * a);
x1_good = c / (a * x2);
```

with

$$a = 1.0,$$
$$b = 200.0,$$
$$c = -1.5 \times 10^{-3}, -1.5 \times 10^{-4}, -1.5 \times 10^{-5}.$$

You will get three sets of solutions:

```
c = 1.0e-3:
    x2              -200.00000
    x1_bad          -7.6293945e-006
    x1_good         -7.5000003e-006

c = 1.0e-4:
    x2              -200.00000
    x1_bad           0.00000000
    x1_good         -7.5000003e-007

c = 1.0e-5:
    x2              -200.00000
    x1_bad           0.00000000
    x1_good         -7.4999996e-008
```

One can verify that `x1_good` should be the correct results. Since `float` holds only 7 decimal digits after the decimal point, subtraction of two similar numbers (i.e., $-b + \Delta$) results in the loss of significance.

Example 2: We want to compute

$$\frac{1 - \cos^2(x)}{x^2}$$

at $x = 10^{-3}$. Since x is small, $\cos(x) \approx 1.0$. To avoid the loss of significance caused by subtraction of two similar numbers, it is recommended to use

$$\frac{\sin^2(x)}{x^2}.$$

If we run the following code fragment, we will see that $f_1 = 0.95367402$ and $f_2 = 0.99999976$, with f_2 being the better solution.

```
float x = 1.0e-3, cos_x, sin_x, f1, f2;
cos_x = cos(x);
sin_x = sin(x);
f1 = (1.0 - cos_x * cos_x)/(x * x);
f2 = sin_x * sin_x /  (x * x);
```

Readers may not be convinced by the above examples, as they may argue that a higher accuracy is achievable when `double` instead of `float` is used. Then, the following example will clear any doubt.

Example 3: Evaluation of many functions such as $\sin x$, $\cos x$, e^x and so on is often performed via evaluation of a truncated *Taylor series* – provided that these functions have converging series. For example,

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!} + \cdots$$

This Taylor (or Maclaurin) series converges uniformly and absolutely for all values of x. Without thoughtful analysis of numerical stability (a necessary step for a good design), one may implement the evaluation as follows:

```
inline long long Factorial(long long n)
{
   long long Result = 1;
   while (n > 1)
   {
      Result *= n;
      n--;
   }
   return Result;
}

void EvaluateExpX(int n, double x, double &f)
{
   int i;

   f = 1.0;
   for (i=1; i<=n; i += 1)
   {
      f += pow(x, i) / Factorial(i);
   }
}
```

It is noted that `Factorial` is declared as an *inline function*. The `inline` specifier instructs the compiler to insert a copy of the function body into each place the function is called. Using inline functions can make our program faster because they eliminate the overhead associated with function calls, but at the cost of larger code size. In computing the factorial $n!$, a `long long` data type is used to avoid overflow. In programming, an overflow occurs when an arithmetic operation attempts to create a numeric value that is too large to be represented within the available storage range. For a 32bit machine, `long long` is equivalent to `_int64` and has the data range of

$$-9,223,372,036,854,775,808 \quad \text{to} \quad 9,223,372,036,854,775,807.$$

This means n should be less than or equal to 20. When n is larger than 20, we will have an integer overflow. We may use `double` to store the factorial and improve the performance with the following implementation:

```
void EvaluateExpX(int n, double x, double &f)
{
   int      i;
   double   factorial;

   f = factorial = 1.0;
   for (i=1; i<=n; i += 1)
   {
      factorial *= i;
      f += pow(x, i) / factorial;
   }
}
```

This is a big improvement in terms of eliminating integer overflow and boosting performance. However, it is still numerically unstable. If you are not convinced, try to evaluate $x = -25$ and compare your result with the one obtained from a pocket calculator or the C internal function `exp(x)`. I will explain why the above implementation is numerically unstable when x is negative in the next chapter. Further improvement in performance and numerical stability of the above implementation will also be discussed.

Example 4: The angle between two unit vectors $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ may be computed via a *dot-product* of two vectors as follows:

$$\boldsymbol{v}_1 \cdot \boldsymbol{v}_2 = \|\boldsymbol{v}_1\| \times \|\boldsymbol{v}_2\| \cos\theta = \cos\theta \quad \Longrightarrow \quad \theta = \text{acos}(\boldsymbol{v}_1 \cdot \boldsymbol{v}_2)$$

noting that $\|\boldsymbol{v}_1\| = 1$ and $\|\boldsymbol{v}_2\| = 1$. In Cartesian coordinates , $\boldsymbol{v}_1 = (x_1, y_1, z_1)$ and $\boldsymbol{v}_2 = (x_2, y_2, z_2)$. Hence,

$$\boldsymbol{v}_1 \cdot \boldsymbol{v}_2 = x_1 x_2 + y_1 y_2 + z_1 z_2.$$

If we use the above formula to compute θ, we will get noticeable numerical noise when two vectors are almost parallel (i.e., θ is very small). To illustrate this instability issue, let's try the following code fragment:

```
double alpha = 1.7453292519943297e-007;
double theta;
double cos_alpha = cos(alpha);
theta = acos(cos_alpha);
```

We expect to obtain $\theta = \alpha$ with respect to a small tolerance ε. However, $\theta = 1.7441362009524762e{-}007$. A numerically stable way to compute θ is to also compute the *cross product* of two vectors:

$$\|\boldsymbol{v}_1 \times \boldsymbol{v}_2\| = \|\boldsymbol{v}_1\| \, \|\boldsymbol{v}_2\| \sin\theta = \sin\theta,$$

where

$$\boldsymbol{v}_1 \times \boldsymbol{v}_2 = \begin{bmatrix} \boldsymbol{i} & \boldsymbol{j} & \boldsymbol{k} \\ x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \end{bmatrix}$$

with $\boldsymbol{i}, \boldsymbol{j}$, and $\boldsymbol{k}$ being the standard basis vectors that represent, in our case, the x-, y-, and z- axes respectively. Then, the angle is obtained by

$$\theta = \arctan\left(\frac{\sin\theta}{\cos\theta}\right) \quad \text{or} \quad \theta = \text{atan2}(\sin\theta, \cos\theta).$$

We now analyze why it is numerically stable to compute θ via arctan or atan2. Floating-point computation has limited precision. The loss of some precision can be reflected by introducing small change of angle δ. Accordingly, we have

$$\frac{\sin(\theta \pm \delta)}{\cos(\theta \pm \delta)}.$$

From trigonometry it is known that

$$\cos(\theta \pm \delta) = \cos(\theta)\cos(\delta) \mp \sin(\theta)\sin(\delta)$$

$$\sin(\theta \pm \delta) = \sin(\theta)\cos(\delta) \pm \cos(\theta)\sin(\delta)$$

When δ is small (caused by numerical noise), we have $\sin(\delta) \approx 0$ and hence:

$$\cos(\theta + \delta) \approx \cos(\theta)\cos(\delta)$$

$$\sin(\theta + \delta) \approx \sin(\theta)\cos(\delta)$$

Consequently,

$$\frac{\sin(\theta)\cos(\delta)}{\cos(\theta)\cos(\delta)} = \frac{\sin(\theta)}{\cos(\theta)} = \tan\theta.$$

This indicates that we should have a "clean" operand for arctan by introducing both sin and cos in computation since numerical noise has been canceled.

We now show that the function arctan is numerically stable when θ approaches zero. Denoting $\tan\theta$ by x, the condition number of $\arctan(x)$ is

$$C_p \approx \left|\frac{xf'(x)}{f(x)}\right| = \frac{1}{1+x^2}\left|\frac{x}{\arctan(x)}\right|$$

When two vectors are almost parallel, θ is small and $x = \tan\theta \to 0$. Accordingly,

$$\lim_{x\to 0}\frac{x}{\arctan(x)} = \lim_{\theta\to 0}\frac{\tan\theta}{\theta} = \lim_{\theta\to 0}\frac{\sin\theta}{\theta}\cos\theta = \lim_{\theta\to 0}\frac{\sin\theta}{\theta} = 1.$$

Therefore, $C_p = 1$ when x approaches zero, indicating that $\arctan(x)$ is stable when x is very small. The proof of

$$\lim_{\theta\to 0}\frac{\sin\theta}{\theta} = 1$$

is often carried out via the Squeeze Theorem but can also be done via L'Hôpital's Rule as:

$$\lim_{\theta\to 0}\frac{\sin\theta}{\theta} = \lim_{\theta\to 0}\frac{\sin'\theta}{\theta'} = \lim_{\theta\to 0}\frac{\cos\theta}{1} = 1.$$

We now look at why arccos is numerically unstable. Denoting $\cos\theta$ by x, the condition number of $\arccos(x)$ is

$$C_p \approx \left|\frac{xf'(x)}{f(x)}\right| = \left|\frac{-1}{\sqrt{1-x^2}}\right|\left|\frac{x}{\arccos(x)}\right|$$

When θ approaches zero, $x = \cos\theta \to 1$. So

$$\lim_{x\to 1}\frac{x}{\arccos(x)} = \lim_{\theta\to 0}\frac{\cos\theta}{\theta} = \infty.$$

Accordingly $C_p \to \infty$, indicating arccos is ill-conditioned near zero.

2.4 Summary

Although integers provide an exact representation of numeric values, they suffer from two major drawbacks: the inability to represent fractional values and a limited dynamic range. Floating-point arithmetic solves these two problems at the expense of accuracy and speed. For many applications, the benefits of floating-point outweigh the disadvantages. Some of the greatest achievements of the 20th century would not have been possible without the floating-point capabilities of digital computers. The fact that floating-point numbers cannot precisely represent all real numbers, as well as the fact that floating-point operations cannot precisely represent true arithmetic operations, is not fully understood by all software developers. Direct comparison of two floating-point values is a bad operation that leads to many surprising situations. In this chapter, we discussed in general how we should compare two floating-point values. There are many research papers that cover more details about floating-point computation. Readers are encouraged to read these papers.

Many engineering and scientific computations are complex and have to be done via computers. A big problem with floating-point arithmetic is that it does not follow the standard rules of algebra. Normal algebraic rules apply only to infinite precision arithmetic. Complex formulae can suffer from larger errors due to round-off. The loss of accuracy can be substantial if a problem or its data are ill-conditioned, meaning that the correct result is hypersensitive to tiny perturbations in its data. One approach to remove the risk of such loss of accuracy is the designing and analysis of numerically stable algorithms, which is a goal of the branch of mathematics known as numerical analysis. Another approach that can protect against the risk of numerical instabilities is the computation of intermediate values in an algorithm at a higher precision than the final result requires, which can remove or reduce such risk by orders of magnitude. For example, we should always use `double` rather than `float` for our intermediate variable in order to retain the highest possible precision even if the final result does not need that much decimal precision.

Numerical analysis is a rigorous mathematical discipline in which such problems and algorithms for their solution are analyzed in order to establish the condition of a problem or the stability of an algorithm and to gain insight into the design of better and more widely applicable algorithms. Numerical analysis is often taught in engineering schools as a graduate-level course. This book is written for a wide range of engineering students and does not assume a very high level of mathematics. Without going into much detail, we discussed how we can minimize the effect of accuracy problems and presented a few practical examples to draw readers' attention. In the subsequent chapters, we will come across more topics related to floating-point computation and numerical analysis.