Chapter 8

Floating Point Arithmetic

You should be familiar with

- Number systems, primarily binary, hexadecimal, and decimal
- Geometric series
- Euclidean vector norm
- Quadratic equation

In this day and age when we rely on computers for so many things, it seems unreasonable to think they make errors. In fact, when performing arithmetic with real numbers such as 0.3 and 1.67×10^8, there is error when the number is placed in computer memory, and it is called *round-off error*. And worse, the error propagates with arithmetic operations such as addition and division. In engineering and scientific applications, it is often necessary to deal with large-scale matrix operations. These computations must be done with minimal error. The engineer or scientist must be aware that errors will occur, why they occur, and how to minimize them.

Engineering applications deal with the computation of functions like e^x, $\cos x$, and the error function $erf(x) = (2/\sqrt{\pi}) \int_0^x e^{-t^2} dt$. These functions are defined in terms of infinite series, and we must cut off summing terms at some point. This is called *truncation error*. Engineering and science applications often have to deal with very large, sparse, matrices. Such matrices have a small number of nonzero entries, and methods of dealing with them are primarily iterative. An iterative method computes a sequence of approximate solutions $\{x_i\}$ that approaches a solution. Such methods suffer from truncation error.

We discuss the representation of numbers in digital computers and its associated arithmetic. A digital computer stores values using the binary number system, where a number is represented by a string of 0's and 1's. Each binary digit is termed a *bit*. Since digital computers have a finite bit capacity (memory), integers and real numbers are represented by a fixed number of binary bits. We will see that integers can be represented exactly as long as the integer value falls within the fixed number of bits. Floating point numbers are another story. Most such values cannot be represented exactly. We will describe representation systems so that we will understand the problems involved when dealing with both integers and floating point numbers.

8.1 INTEGER REPRESENTATION

Suppose that p bits are available to represent an integer. Here is a simple way to do it. A positive integer has a zero in the last bit and the $p-1$ other bits contain the binary (base-2) representation of the integer. For example, for $p = 8$, the positive integer

$21 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$ is encoded as

0	0	0	1	0	1	0	1

and $58 = 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$ is encoded as

0	0	1	1	1	0	1	0

For negative integers, most computers use the *two's-complement representation*. The system works like an ideal odometer. If the odometer reads 000000 and the car backs up 1 mile, the odometer reads 999999. In binary with $p = 8$, zero is 00000000, so -1 becomes 11111111. Continuing in this fashion, -2 becomes 11111110, -3 is 11111101, and so forth. This might appear strange, but it is really very effective and simple. If this representation of -1 is to make sense, there must be a logical way to take the negative of 1 and obtain this representation for -1. Invert all the bits ($0 \to 1, 1 \to 0$) and add 1.

Numerical Linear Algebra with Applications. http://dx.doi.org/10.1016/B978-0-12-394435-1.00008-9

$$-1 \rightarrow \text{invert}\,(00000001) + 1 = 11111110 + 1 = 11111111. \tag{8.1}$$

The inversion of bits is called the *1s-complement*, which we indicate by 1comp(n), so we can write Equation 8.1 as

$$-1 \rightarrow 1\text{comp}(0000001) + 1.$$

1	1	1	1	1	1	1	1

The negative of 3 should be 11111101. To verify this, compute

$$-3 \rightarrow 1\text{comp}(00000011) + 1 = 11111100 + 1 = 11111101.$$

1	1	1	1	1	1	0	1

Negation works both ways. For instance, $-(-2) = 2$.

$$-(-2) = 1\text{comp}(11111110) + 1 = 00000001 + 1 = 00000010.$$

0	0	0	0	0	0	1	0

Definition 8.1. We denote the process of taking the one's complement and adding 1 with the notation 2comp(n). Thus, the negative of a two's complement number n is $2\text{comp}(n) = 1\text{comp}(n) + 1$.

Remark 8.1. The leftmost bit is called the *sign bit*. It is always 0 when the integer is positive and 1 when it is negative.

We still have not discussed how to add or subtract two's-complement numbers. If we add the representations for -1 and 1 together, we should get 0. Thus, form the sum using ordinary binary arithmetic:

$$11111111 + 00000001 = \underline{1}|00000000,$$

where the underlined bit is the *carry*. Discard the carry, retaining 8 bits, and we have a result of 00000000, or zero. One more example will suggest a formula for addition of two's complement numbers.

Example 8.1. Form the sum of 95 and -43.

$$95 \rightarrow 01011111$$
$$43 \rightarrow 00101011, \quad \text{and} \quad -43 \rightarrow 11010100 + 1 = 11010101$$
$$95 + (-43) \rightarrow 01011111 + 11010101 = \underline{1}|00110100$$

Discard the carry, and the result is $00110100 \rightarrow 52$.

0	0	1	1	0	1	0	0

■

Remark 8.2. The following rules show how to perform addition or subtraction of two's-complement numbers.

- Given two p-bit integers m and n, form $m + n$ by performing binary addition and discarding the carry.
- The subtraction $m - n$ is performed adding $(-n)$ to m, so $m - n = m + 2\text{comp}(n)$.

The largest positive integer is $0\underbrace{111\ldots111}$ $= 2^{p-1} - 1$. Finding the most negative integer is more interesting. The representation of -1 is

1	1	1	1	1	1	1	1

and the representation of -2 is

1	1	1	1	1	1	1	0

Continuing in this way, we eventually arrive at

1	0	0	0	0	0	0	0

Going from 1111111 to 10000001 comprises $127 = 2^7 = 2^{p-1} - 1$ integers, and the range is $-1 \geq n \geq -127$. We get one more negative integer, namely, $1\underbrace{000\ldots000}_{p-1}$ that represents -2^{p-1}, so the range of integers we can represent is $-2^{p-1} \leq n \leq 2^{p-1} - 1$.

Example 8.2. You have heard of 32-bit and 64-bit operating systems. In a 32-bit system, $p = 32$, so the range of integers that can be represented is

$-2^{31} \leq n \leq 2^{31} - 1$ or $-2,147,483,648 \leq n \leq 2,147,483,647$.

In a 64-bit system, $p = 64$, and the range of an integer is

$-18,446,744,073,709,551,616 \leq n \leq 18,446,744,073,709,551,615$. ■

Integer arithmetic can cause *overflow* when the range of a positive or negative integer is exceeded. For instance, let's assume $p = 8$ and add 120 and 88. Then

0	1	1	1	1	0	0	0	+	0	1	0	1	1	0	0	0	=	1	1	0	1	0	0	0	0

which is the value -48.

Remark 8.3. When the sign of the result is the opposite of what it should be, overflow has occurred.

Example 8.3. Compute $-18 + (-112) = 11101110 + 10010000 = \underline{1}|01111110$, so $-18 + (-112) = 126$. This results is an overflow. ■

During integer operations, all results must lie within the interval $-2^{p-1} \leq n \leq 2^{p-1} - 1$. This is a significant limitation. If large integer arithmetic must be performed, software is required, often called a BigInteger package. For example, data encryption usually requires operations with very large integers.

8.2 FLOATING-POINT REPRESENTATION

For given integers b, p, e_{min}, and e_{max}, we define the nonzero floating-point numbers as real numbers of the form

$$\pm \left(0.d_1 d_2 \ldots d_p\right) \times b^n$$

with $d_1 \neq 0$, $0 \leq d_i \leq b - 1$, and $-e_{min} \leq n \leq e_{max}$. We denote by F the (finite) set of all floating-point numbers. In this notation,

a. b is the base. The most common bases are $b = 2$ (binary base), $b = 10$ (decimal base), and $b = 16$ (hexadecimal base).
b. $-e_{min} \leq n \leq e_{max}$ is the exponent that defines the order of magnitude of the number to be encoded.
c. The integers $0 \leq d_i \leq b - 1$ are called the digits and p is the number of *significant digits*. The *mantissa* is the integer $m = d_1 d_2 \ldots d_p$. Note that

$$\left(0.d_1 d_2 \ldots d_p\right) \times b^n = \left(d_1 \times b^{-1} + d_2 b^{-2} + \cdots + d_p b^{-p}\right) b^n = b^n \sum_{k=1}^{p} d_k b^{-k}. \tag{8.2}$$

The following bounds hold for floating-point numbers

$$f_{min} \leq |f| \leq f_{max} \quad \text{for every } f \in F,$$

where $f_{min} = b^{-(e_{min}+1)}$ is the smallest positive real number. We can see this by setting $n = -e_{min}, d_1 = 1, d_2 = d_3 = \cdots = d_p = 0$ and applying Equation 8.2. The largest floating point number f_{max} is found by applying Equation 8.2 with $d_i = (b-1)$, $1 \leq i \leq p$ and exponent $n = e_{max}$. Using the formula for the sum of a geometric series

$$f_{max} = b^{e_{max}} (b-1) \left(b^{-1} + b^{-2} + \cdots + b^{-p}\right) = b^{e_{max}} (b-1) \left(\frac{1 - \left(\frac{1}{b}\right)^p}{b-1}\right)$$

$$= b^{e_{max}} \left(1 - \left(\frac{1}{b}\right)^p\right) = b^{e_{max}} \left(1 - b^{-p}\right).$$

Summary

$$f_{\min} = b^{-(e_{\min}+1)}$$
$$f_{\max} = b^{e_{\max}}\left(1 - b^{-p}\right)$$

Smaller numbers produce an *underflow* and larger ones an *overflow*.

Computers use base $b = 2$ and usually support single precision (representation with $p = 32$) and double precision (representation with $p = 64$). In our single-precision representation, we use 1 bit for the sign (0 means positive, 1 means negative), 8 bits for the exponent, and 23 bits for the mantissa (for a total of 32 bits). In our double-precision representation, we use 1 bit for the sign, 11 bits for the exponent, and 52 bits for the mantissa (for a total of 64 bits). For example, represent 81.625 in single precision.

$$\begin{aligned} 81.625 &= (1)\,2^6 + (0)\,2^5 + (1)\,2^4 + (0)\,2^3 + (0)\,2^2 + (0)\,2^1 + (1)\,2^0 + (1)\,2^{-1} + (0)\,2^{-2} + (1)\,2^{-3} \\ &= \left((1)\,2^{-1} + (0)\,2^{-2} + (1)\,2^{-3} + (0)\,2^{-4} + (0)\,2^{-5} + (0)\,2^{-6} + (1)\,2^{-7} + (1)\,2^{-8} + (0)\,2^{-9} + (1)\,2^{-10}\right)2^7 \end{aligned}$$

Note that the leading digit is $d_1 = 1$, as required, and the exponent is properly adjusted. To avoid dealing with a negative exponent, encode it as an unsigned integer by adding to it a "bias" (127 is the usual bias in single precision). For our example of 81.625, the exponent will be stored as $7 + 127 = 134$ using bias 127. To obtain the actual exponent, compute $134 - 127 = 7$. If the exponent is -57, it is stored as $-57 + 127 = 70$. A stored exponent of 0 reflects an actual exponent of -127. Here is the internal representation for 81.625, where sign $= 0$, exponent $= 134$ in a field of 8 bits, and the mantissa follows the exponent in a field of 23 bits.

0|10000110|10100011010000000000000

There still remains the issue of $x = 0.0$. Represent it by filling both the exponent and mantissa with zeros. The sign bit can still be 0 (+) or 1 (−), so you will sometimes see output like −0.0000. Note that this situation does not occur with the two's-complement integer representation.

Example 8.4. Let $x = \frac{1}{6}$, and assume that $b = 2$ and $p = 8$. The binary representation of 1/6 is the infinite repeating pattern

$$0.00101010101010101010101010101010\ldots$$

We only have eight binary digits to work with, so when 1/6 is entered into computer memory, the binary sequence is either rounded or truncated. When using *rounding*, the approximation is 0.00101011, but with *truncation* the approximation is 0.00101010. ■

To indicate the error in converting a floating number, x, to its computer representation, we use the notation $fl\,(x)$.

Definition 8.2. Let fl (x) be the floating-point number associated with the real number x. For instance, in Example 8.4, the number $x = \frac{1}{6}$ was approximated using rounding, so

$$\text{fl}\,(x) = 0.00101011.$$

8.2.1 Mapping from Real Numbers to Floating-Point Numbers

The most widely used standard for floating-point computation is the *IEEE Standard for Floating-Point Arithmetic*. The most frequently used IEEE formats are single and double precision. In each case, a nonzero number is assumed to have a hidden 1 prior to the first digit. As a result, single precision uses 24 binary digits, and double precision 53. Table 8.1 lists the attributes of the two formats.

TABLE 8.1 IEEE Formats

Name	Base	Digits	e_{min}	e_{max}	Approximate Decimal Range
Single precision	2	23+1	−126	+127	1.18×10^{-38} *to* 3.4×10^{38}
Double precision	2	52+1	−1022	+1023	2.23×10^{-308} *to* 1.80×10^{308}

Floating point numbers are *granular*, which means there are gaps between numbers. The granularity is caused by the fact that a finite number of bits are used to represent a floating point number. We represented 81.625 perfectly because 0.625 is $\frac{1}{2} + \frac{1}{8}$, but most real numbers must be approximated because there is no exact conversion into binary (Example 8.4).

The distance from 1.0 to the next largest double-precision number is 2^{-52} in IEEE double precision. If a number smaller than 2^{-52} is added to 1, the result will be 1. The floating point numbers between 1.0 and 2.0 are equally spaced:

$$\left\{1, 1 + 2^{-52}, 1 + 2 \times 2^{-52}, 1 + 3 \times 2^{-52}, \ldots, 2\right\}.$$

The gap increases as the length of intervals become larger by a factor of 2. The numbers between 2.0 and 4.0 are separated by a gap of 2^{-51}.

$$2\left\{1, 1 + 2^{-52}, 1 + 2 \times 2^{-52}, 1 + 3 \times 2^{-52}, \ldots, 2\right\}.$$

In general, an interval from 2^k to 2^{k+1} has a gap between values of $2^k\left(2^{-52}\right)$. As k increases, the gap relative to 2^k remains 2^{-52}. In single precision, the relative gap between numbers is 2^{-23}. There is a name associated with this gap. It is called the *machine precision*, or eps, and it plays a significant role in analysis of floating point operations. Remember that the value of eps varies with the precision.

There is a formula for eps for any b and p, and let's intuitively determine it. Let $b = 10$ and $p = 4$, and assume we round to p digits. Let $x = 1$. Then, $\text{fl}\,(1 + 0.0001) = 1, \text{fl}\,(1 + 0.0003) = 1, \text{fl}\,(1 + 0.0004) = 1, \text{fl}\,(1 + 0.0005) = 1.0001$. If we repeat the experiment with $b = 10$, $p = 5$, we will find that $\text{fl}\,(1 + 0.00005) = 1.00001$. Let $b = 2, p = 3$ and again assume rounding. Let $x = 1$. Now, $\text{fl}\left(1 + 2^{-5}\right) = \text{fl}\,(1 + 0.00001) = 1$, $\text{fl}\left(1 + 2^{-4}\right) = \text{fl}\,(1 + 0.0001) = 1$, $\text{fl}\left(1 + 2^{-3}\right) = \text{fl}\,(1 + 0.001) = 1, \text{fl}(1 + 2^{-2}) = \text{fl}(1.01) = 1.01$. This is enough information for us to define eps.

Definition 8.3. Assume b is the base of the number system, p is the number of significant digits, and that rounding is used. The machine precision, $\text{eps} = \frac{1}{2}b^{1-p}$, is the distance from 1.0 to the next largest floating point number.

For IEEE double-precision floating-point, we specified the $\text{eps} = 2^{-52}$. Applying the formula with $b = 2$ and $p = 52$, we have $\text{eps} = \frac{1}{2}2^{1-52} = 2^{-52}$. In single precision, $\text{eps} = \frac{1}{2}2^{1-23} = 2^{-23}$. Figure 8.1 shows the distribution of a floating-point number system with $b = 2, p = 3, \text{e}_{\min} = -3, \text{e}_{\max} = 3$, so $\text{eps} = \frac{1}{2}2^{-2} = \frac{1}{8}$. Notice how the gaps between numbers grow, but remember the gap remains constant relative to the number size. The nonnegative floating point numbers shown in Figure 8.1 are:

```
0.0000    0.0625   0.078125   0.09375   0.109375
 0.125   0.15625     0.1875    0.21875
  0.25    0.3125      0.375     0.4375
   0.5     0.625       0.75      0.875
1.0000    1.2500     1.5000    1.7500
2.0000    2.5000     3.0000    3.5000
4.0000    5.0000     6.0000    7.0000
```

The conversion of real numbers to floating-point numbers is called floating-point representation or *rounding*, and the error between the true value and the floating-point value is called *round-off error*. We expect $(fl\,(x) - x)/x = \epsilon$ not to exceed eps in magnitude. The following formula holds for all real numbers $f_{\min} \leq x \leq f_{\max}$:

$$fl\,(x) = x\,(1 + \varepsilon) \tag{8.3}$$

with $|\epsilon| \leq \text{eps}$.

Remark 8.4. For single-precision IEEE floating point representation, $\text{eps} = 2^{-23} \approx 1.192 \times 10^{-7}$, and for double precision, $\text{eps} = 2^{-52} \approx 2.22 \times 10^{-16}$. These numbers explain the maximum accuracy of 7 or 16 significant digits for single or double-precision arithmetic. Figure 8.2 shows a map of double-precision IEEE floating-point numbers. Figures 8.1 and 8.2 together provide a good picture of a floating-point number system.

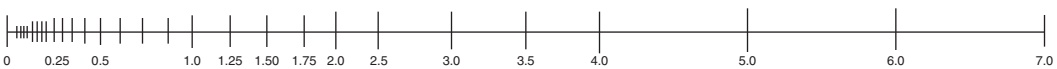

FIGURE 8.1 Floating-point number system.

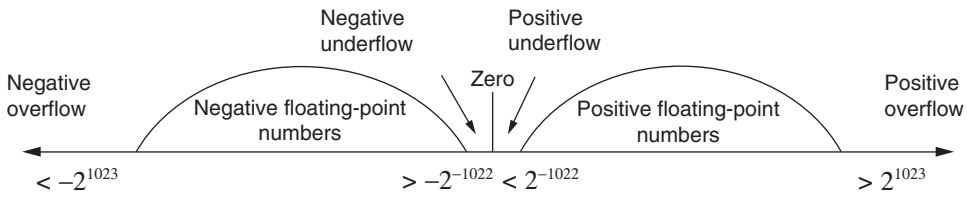

FIGURE 8.2 Map of IEEE double-precision floating-point.

8.3 FLOATING-POINT ARITHMETIC

Consider the operation $+$. The sum of two floating-point numbers is usually an approximation to the actual sum. We denote by $\oplus$ the computer result of the addition.

Definition 8.4. For real numbers x and y,

$$x \oplus y = \text{fl}\,(fl\,(x) + \text{fl}\,(y))\,. \tag{8.4}$$

Overflow occurs if the addition produces a number that is too large, $|x \oplus y| > f_{\max}$, and underflow occurs if it produces a number that is too small, $|x \oplus y| < f_{\min}$. Similar notation applies for the other operations: $\ominus$, $\otimes$, and $\oslash$.

Example 8.5. Assume $b = 10, p = 4$, and that the true values of x and y are 0.34578×10^1 and 0.56891×10^1, respectively,

$$\text{fl}\,(x) = 0.3458 \times 10^1, \quad \text{fl}\,(y) = 0.5689 \times 10^1,$$

and

$$x \oplus y = 0.9147 \times 10^1.$$ ■

When performing floating point operations on values with different exponents, a realignment must take place. For instance, let $b = 10, p = 5, x = 0.10002 \times 10^2$ and $y = 0.99982 \times 10^1$. Now compute $x \ominus y$.

0	.	1	0	0	0	2	**0**	×	10^2
				−					
0	.	0	9	9	9	8	2	×	10^2
				=					
0	.	3	8	0	0	0		×	10^{-2}

Without adding the extra 0 in x and sticking with 5 digits throughout the calculation, we will get

1	0	.	0	0	2
		−			
	9	.	9	9	8
		=			
0	.	0	0	4	0

an incorrect result. The additional zero is called a *guard digit.* Most computers use guard digits, so we will assume that all calculations are done using them.

Remark 8.5. Floating point numbers have a fixed range so, as is the case with integers, dealing with floating point numbers with many significant digits and large exponents requires software.

8.3.1 Relative Error

There are two ways to measure error, using *absolute error* or *relative error*.

$$\begin{aligned} \text{Absolute error} &= |\text{fl}\,(x) - x| \\ \text{Relative error} &= \frac{|\text{fl}\,(x) - x|}{|x|}, \quad x \neq 0 \end{aligned}$$

Example 8.6.

a. In Example 8.5, $b = 10$ and $p = 4$. The value of eps for this representation is $\text{eps} = 0.0005 = 5 \times 10^{-4}$. The value $x = 0.34578 \times 10^1$ converts to floating point as $\text{fl}(x) = 0.3458 \times 10^1$. According to Equation 8.3, $\text{fl}\left(0.34578 \times 10^1\right) = 0.3458 \times 10^1 = 0.34578 \times 10^1 (1 + \varepsilon)$, so $\epsilon = ((0.3458 \times 10^1)/(0.34578 \times 10^1)) - 1 = 0.5784 \times 10^{-4} < \text{eps}$, as expected. Also, $|\text{fl}(x) - x| = 0.0002$, as opposed to $(|fl(x) - x|)/|x| = 0.5784 \times 10^{-4}$.

b. Consider $x = 1.6553 \times 10^5$, $\text{fl}(x) = 1.6552 \times 10^5$. The absolute error is $|\text{fl}(x) - x| = 10$, while the relative error is $(|\text{fl}(x) - x|)/|x| = 6.04 \times 10^{-5}$. With large numbers, relative error is generally more meaningful, as we see here. This same type of example applies to small values.

c. Relative error gives an indication of how good a measurement is relative to the size of the thing being measured. Let's say that two students measure the distance to different objects using triangulation. One student obtains a value of $d_1 = 28.635\,\text{m}$, and the true distance is $\overline{d_1} = 28.634\,\text{m}$. The other student determines the distance is $d_2 = 67.986\,\text{m}$, and the true distance is $\overline{d_2} = 67.987\,\text{m}$. In each case, the absolute error is 0.001. The relative errors are $(|28.634 - 28.635|)/|28.634| = 3.49 \times 10^{-5}$ and $(|67.987 - 67.986|)/|67.987| = 1.47 \times 10^{-5}$. The relative error of measurement d_2 is about 237% better than that of measurement d_1, even though the amount of absolute error is the same in each case. ■

Relative error provides a much better measure of change for almost any purpose. For instance, estimate the sum of the series

$$\sum_{i=1}^{\infty} \frac{1}{i^2 + \sqrt{i}}.$$

Compute the sequence of partial sums $s_n = \sum_{i=1}^{n} 1/(i^2 + \sqrt{i})$. until a given error tolerance, tol, is attained. The actual sum of the series is not known, so a comparison of the partial sum with the actual sum cannot be computed. There are two approaches commonly used:

a. Compute partial sums until $|s_{n+1} - s_n| < \text{tol}$.

b. Compute partial sums until $\frac{|s_{n+1} - s_n|}{|s_n|} < \text{tol}$.

Method 2 is preferable because it tells us how the new partial sum is changing relative to the previous sum.

Remark 8.6. Most computer implementations of addition (including the widely used IEEE arithmetic) satisfy the property that the relative error is less than the machine precision:

$$\left| \frac{(x \oplus y) - (x + y)}{x + y} \right| \leq \text{eps}$$

assuming $x + y \neq 0$.

The relative error for one operation is very small, but this is not always the case when a computation involves a sequence of many operations.

8.3.2 Rounding Error Bounds

It is important to understand how floating point errors propagate, since this leads to means of controlling them. We will do a mathematical analysis of rounding error for the addition of floating point numbers, but will not do so for $\ominus$, $\otimes$, or $\oslash$, or error propagation of vector and matrix operations. We will state results, and the interested reader can consult Refs. [9, 16, 17] for a rigorous analysis. In all cases, we will assume that the approximation of x by $\text{fl}(x)$ is done by rounding rather than truncation.

Remark 8.7. IEEE 754 requires that arithmetic operations produce results that are exactly rounded, i.e., the same as if the values were computed to infinite precision prior to rounding.

We will assume that once floating point numbers x and y are in computer memory that the basic arithmetic operations satisfy the following:

For all floating point numbers x, y in a computer:

$$\begin{aligned} x \oplus y &= (x+y)(1+\epsilon), \\ x \ominus y &= (x-y)(1+\epsilon), \\ x \otimes y &= (x \times y)(1+\epsilon), \\ x \oslash y &= (x/y)(1+\epsilon), \end{aligned} \tag{8.5}$$

where $|\epsilon| \leq \text{eps}$.

If the reader is not interested in the technical details, the results for addition and multiplication of floating point numbers can be summarized as follows:

When adding n floating point numbers, the result is the exact sum of the n numbers, each perturbed by a small relative error. The errors are bounded by $(n-1)$ eps, where eps is the unit roundoff error.

The relative error in computing the product of n floating point numbers is at most $1.06\,(n-1)$ eps, assuming that $(n-1)\,\text{eps} < 0.1$.

There are error bounds for matrix operations that depend on eps and the magnitude of the true values.

Addition

Lemma 8.1. *Let $x_1, x_2, \ldots, x_n$ be positive floating point numbers in a computer. Then,*

$$\begin{aligned} x_1 \oplus x_2 \oplus x_3 \oplus \cdots \oplus x_n &= x_1(1+\varepsilon_1)(1+\varepsilon_2)\ldots(1+\epsilon_{n-1}) \\ &\quad + x_2(1+\varepsilon_1)(1+\varepsilon_2)\ldots(1+\epsilon_{n-1}) \\ &\quad + x_3(1+\varepsilon_2)(1+\varepsilon_3)\ldots(1+\epsilon_{n-1}) \\ &\quad \vdots \\ &\quad + x_{n-1}(1+\epsilon_{n-2})(1+\varepsilon_{n-1}) \\ &\quad + x_n(1+\epsilon_{n-1}), \end{aligned}$$

where $|\epsilon_i| \leq \text{eps}$, $1 \leq i \leq n-1$.

Proof. Assume the computation proceeds as follows:

$$s_2 = x_1 \oplus x_2, s_3 = s_2 \oplus x_3, \ldots, s_n = s_{n-1} \oplus x_n.$$

From Equation 8.5,

$$s_2 = \text{fl}(x_1 + x_2) = (x_1 + x_2)(1+\epsilon_1) = x_1(1+\epsilon_1) + x_2(1+\epsilon_1),$$

where $|\epsilon_1| \leq \text{eps}$. Now,

$$\begin{aligned} s_3 &= fl(s_2 + x_3) = (s_2 + x_3)(1+\epsilon_2) \\ &= s_2(1+\epsilon_2) + x_3(1+\epsilon_2) \\ &= x_1(1+\epsilon_1)(1+\epsilon_2) \\ &\quad + x_2(1+\varepsilon_1)(1+\epsilon_2) \\ &\quad + x_3(1+\varepsilon_2). \end{aligned} \tag{8.6}$$

Equation 8.6 defines a pattern, and we have

$$\begin{aligned} s_n &= x_1 (1+\epsilon_1)(1+\epsilon_2)\dots(1+\epsilon_{n-1}) \\ &\quad +x_2 (1+\epsilon_1)(1+\epsilon_2)\dots(1+\epsilon_{n-1}) \\ &\quad +x_3 (1+\epsilon_2)(1+\epsilon_3)\dots(1+\epsilon_{n-1}) \\ &\quad \vdots \\ &\quad +x_{n-1}(1+\varepsilon_{n-2})(1+\varepsilon_{n-1}) \\ &\quad +x_n (1+\varepsilon_{n-1}) \end{aligned} \tag{8.7}$$

where $|\epsilon_i| \leq \text{eps}, 1 \leq i \leq n-1$. □

Following the analysis in Ref. [17, pp. 132-134], define

$$\begin{aligned} 1+\eta_1 &= (1+\epsilon_1)(1+\epsilon_2)\dots(1+\epsilon_{n-1}), \\ 1+\eta_2 &= (1+\epsilon_1)(1+\epsilon_2)\dots(1+\epsilon_{n-1}), \\ 1+\eta_3 &= (1+\epsilon_2)(1+\epsilon_3)\dots(1+\epsilon_{n-1}), \\ &\vdots \\ 1+\eta_{n-1} &= (1+\varepsilon_{n-2})(1+\varepsilon_{n-1}), \\ 1+\eta_n &= (1+\varepsilon_{n-1}). \end{aligned}$$

We can now write Equation 8.7 as

$$\begin{aligned} s_n &= x_1(1+\eta_1) + x_2(1+\eta_2) + x_3(1+\eta_3) + \dots + \\ &\quad x_{n-1}(1+\eta_{n-1}) + x_n(1+\eta_n). \end{aligned} \tag{8.8}$$

In order for Equation 8.8 to be useful, we need bounds for the η_i. From $1+\eta_n = (1+\varepsilon_{n-1})$, it follows that $|\eta_n| = |\epsilon_n| \leq \text{eps}$. Now consider the term $1+\eta_{n-1}$:

$$1+\eta_{n-1} = 1+\epsilon_{n-1}+\epsilon_{n-2}+\epsilon_{n-1}\epsilon_{n-2},$$

so

$$\eta_{n-1} = \epsilon_{n-1}+\epsilon_{n-2}+\epsilon_{n-1}\epsilon_{n-2},$$

and

$$|\eta_{n-1}| \leq |\epsilon_{n-1}+\epsilon_{n-2}| + |\epsilon_{n-1}\epsilon_{n-2}| \leq |\epsilon_{n-1}| + |\epsilon_{n-2}| + |\epsilon_{n-1}\epsilon_{n-2}|.$$

The term $|\epsilon_{n-1}\epsilon_{n-2}|$ is bounded by 2eps^2. If we are using double-precision arithmetic, $\text{eps} = 2^{-52}$, so $2\text{eps}^2 = 2^{-103}$ and we can consider $|\epsilon_{n-1}\epsilon_{n-2}|$ negligible compared to $|\epsilon_{n-1}| + |\epsilon_{n-2}|$. Thus,

$$|\eta_{n-1}| \leq |\epsilon_{n-1}| + |\epsilon_{n-2}| \leq 2\,\text{eps}.$$

Continuing in this fashion, we will obtain

$$\begin{aligned} |\eta_1| &\leq (n-1)\,\text{eps}, \\ |\eta_i| &\leq (n-i+1)\,\text{eps}, \quad 2 \leq i \leq n. \end{aligned} \tag{8.9}$$

Equations 8.8 and 8.9 can be summarized as follows:

> When adding n floating point numbers, the result is the exact sum of the n numbers, each perturbed by a small relative error. The errors are bounded by $(n-1)$ eps, where eps is the unit roundoff error.

It is useful to derive a bound for the relative error. Let

$$s = \sum_{i=1}^{n} x_i \tag{8.10}$$

Subtract Equation 8.10 from Equation 8.7 to obtain

$$s_n - s = x_1\eta_1 + x_2\eta_2 + \cdots + x_n\eta_n. \tag{8.11}$$

Equation 8.9 implies that

$$|\eta_i| \leq (n-1)\,\text{eps}, \quad 1 \leq i \leq n. \tag{8.12}$$

Taking the absolute value of both sides of Equation 8.11 and applying 8.12, we have

$$|s_n - s| \leq (|x_1| + |x_2| + \cdots + |x_n|)\,(n-1)\,\text{eps},$$

This result leads to Theorem 8.1.

Theorem 8.1. *If n floating point numbers, x_i, are added,*

$$\frac{|s_n - s|}{|x_1 + x_2 + \cdots + x_n|} \leq K\,(n-1)\,\text{eps},$$

where

$$K = \frac{(|x_1| + |x_2| + \cdots + |x_n|)}{|x_1 + x_2 + \cdots + x_n|}.$$

Multiplication

Theorem 8.2. *The relative error in computing the product of n floating point numbers is at most* $1.06\,(n-1)\,\text{eps}$, *assuming that* $(n-1)\,\text{eps} < 0.1$.

Matrix Operations

Theorem 8.3. *For an $m \times n$ matrix M, define $|M| = \left(\left|m_{ij}\right|\right)$; that is $|M|$ is the matrix whose entries are the absolute value of those from M. Let A and B be two floating point matrices and let c be a floating point number. Then*

$$\text{fl}\,(cA) = cA + E, \quad |E| \leq \text{eps}\,|cA|$$

$$\text{fl}\,(A + B) = (A + B) + E, \quad |E| \leq \text{eps}\,|A + B|\,.$$

If the product of A and B is defined, then

$$\text{fl}\,(AB) = AB + |E|\,, \quad |E| \leq n\ \text{eps}\,|A|\,|B| + K\text{eps}^2,$$

where K is a constant.

Example 8.7. Assume $b = 10$ and $p = 5$. This example will test the error bound asserted for addition in Theorem 8.1. With this representation, eps $= 0.00005$. Table 8.2 gives the numbers, $\overline{x_i}$, before they are entered into the computer and the floating point approximations $\text{fl}\,(\overline{x_i}) = x_i$.

TABLE 8.2 Floating-Point Addition

$\bar{x}$	0.562937×10^0	0.129873×10^1	0.453219×10^1	0.765100×10^0	0.120055×10^1
x	0.56294×10^0	0.12987×10^1	0.45322×10^1	0.76510×10^0	0.12006×10^1

$$s = \sum_{i=1}^{5} x_i = 0.835954 \times 10^1$$

$$s_2 = fl\,(x_1 + x_2) = 0.18616 \times 10^1,\ s_3 = fl\left(0.18616 \times 10^1 + x_3\right) = 0.63938 \times 10^1.$$

$$s_4 = fl\left(0.63938 \times 10^1 + x_4\right) = 0.71589 \times 10^1,\ s_5 = fl\left(0.71589 \times 10^1 + x_5\right) = 0.83595 \times 10^1.$$

Now,

$$\frac{|s_5 - s|}{|s|} = \frac{\left|0.83640 \times 10^1 - 0.835954 \times 10^1\right|}{\left|0.835954 \times 10^1\right|} = \frac{.40000 \times 10^{-4}}{0.835954 \times 10^1} = 0.47850 \times 10^{-5}.$$

The value of K in Theorem 8.1

$$K = \frac{|x_1| + |x_2| + |x_3| + |x_4| + |x_5|}{|x_1 + x_2 + x_3 + x_4 + x_5|} = 1,$$

and

$$K\,(n-1)\,eps = (1)\,(4)\,(.00005) = 2.0000 \times 10^{-4},$$

validating the error bound of Theorem 8.1.

■

8.4 MINIMIZING ERRORS

The subject of error minimization is complex and sometimes involves clever tricks, but there are some general principles to follow. The article, *What Every Computer Scientist Should Know About Floating Point Arithmetic* [18], is an excellent summation of earlier sections of this chapter and contains a discussion of some techniques for minimizing errors.

8.4.1 Avoid Adding a Huge Number to a Small Number

Adding a very large number to a small number may eliminate any contribution of the small number to the final result.

Example 8.8. Assume $b = 10, p = 4$. Let $x = 0.267365 \times 10^3$ and $y = 0.45539 \times 10^{-3}$. Then,

$$\text{fl}\,(x) = 0.2674 \times 10^3, \quad \text{fl}\,(y) = 0.4554 \times 10^{-2},$$

and

$$\text{fl}\,(fl\,(x) + \text{fl}\,(y)) = 0.2674 \times 10^3.$$

■

Overflow can result from including very large numbers in calculations, and can sometimes be avoided by just modifying the order in which you do the computation. For instance, suppose you need to compute the Euclidean norm of a vector x:

$$\|x\|_2 = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}$$

If some x_i are very large, overflow can occur. There is a way to avoid this. Compute $m = \max\left(|x_2|, |x_2|, \ldots, |x_n|\right)$, divide each x_i by m (called *normalizing* the vector), and then sum the squares of the normalized vector components and multiply by m. Overflow will be avoided. Here are the steps:

a. $m = \max\left(|x_2|, |x_2|, \ldots, |x_n|\right)$.
b. $y_i = x_i/m,\ 1 \le i \le n\left(y_i^2 \le 1\right)$.
c. $\|x\|_2 = m\sqrt{y_1^2 + y_2^2 + \cdots + y_n^2}$.

8.4.2 Avoid Subtracting Numbers That Are Close

Solving the quadratic equation

$$ax^2 + bx + c = 0, \quad a \neq 0$$

is a classic example where subtracting numbers that are close can be disastrous. We call this *cancellation* error. The usual way to finds its two roots is by using the *quadratic formula*:

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

If the product $4ac$ is small, then $\sqrt{b^2 - 4ac} \cong b$, and when computing one of x_1 or x_2 we will be subtracting two nearly equal numbers. This can be a serious problem, since we know a computer maintains only a fixed number of significant digits.

Example 8.9. Consider solving $ax^2 + bx + c = 0$ with $a = 1$, $b = 68.50$, $c = 0.1$ with the quadratic equation. Use rounded base-10 arithmetic with 4 significant digits. Now, $b^2 - 4ac = 4692 - 0.4000 = 4692$, and

$$x_1 = \frac{-68.50 + \sqrt{4692}}{2} = \frac{-68.50 + 68.50}{2} = 0,$$
$$x_2 = \frac{-68.50 - \sqrt{4692}}{2} = \frac{-68.50 - 68.50}{2} = -68.50.$$

The correct roots are

$$x_1 = -0.001460,$$
$$x_2 = -68.50.$$

The relative error in computing x_1 is $(|-0.001460 - 0.0000|)/|-0.0001460| = 1.0000$, which is quite awful; however, x_2 is correct. There are two causes of the problem. First, the contribution of $-4ac$ was lost during the subtraction from a much larger number, followed by the cancellation error. Cancellation can be avoided by writing the quadratic formula in a different way.

Multiply the two solutions:

$$\left(\frac{-b + \sqrt{b^2 - 4ac}}{2a}\right)\left(\frac{-b - \sqrt{b^2 - 4ac}}{2a}\right) = \frac{b^2 - \left(b^2 - 4ac\right)}{4a^2} = \frac{4ac}{4a^2} = \frac{c}{a}.$$

Thus, $x_1 x_2 = c/a$. Pick the one of the two solutions that does not cause subtraction and call it x_1.

$$x_1 = -\left(\frac{b + \text{sign}\,(b)\,\sqrt{b^2 - 4ac}}{2a}\right),$$

where sign (b) is $+1$ if $b > 0$ and -1 if $b < 0$. Then compute x_2 using

$$x_2 = \frac{c}{ax_1}.$$

Cancellation is avoided. For our example,

$$x_1 = -\left(\frac{68.5 + 68.5}{2}\right) = -68.5,$$
$$x_2 = \frac{0.1}{(1)\,(-68.5)} = -0.001460.$$ ■

There are other classic examples where cancellation error causes serious problems, and some are included in the problems. See Ref. [19, pp. 42-44] for some interesting examples.

We will conclude this section by saying that underflow and overflow are not the only errors produced by a floating-point representation. The MATLAB constant `inf` returns the IEEE arithmetic representation for positive infinity, and in some situations its use is valid. Infinity is also produced by operations like dividing by zero (`1.0/0.0`), or from overflow (`exp(750)`). NaN is the IEEE arithmetic representation for Not-a-Number. A NaN results from mathematically undefined operations such as `0.0/0.0` and `inf-inf`. In practice, obtaining a `NaN` or an unexpected `inf` is a clear indication that something is wrong!

8.5 CHAPTER SUMMARY

Representation of Integers

Integers are represented using two's-complement notation. If the hardware uses n bits to store an integer, then the left-most bit is the sign bit, and is 0 if the integer is nonnegative and 1 if it is negative. The system functions like an ideal odometer. The representation $000\ldots00$ is the integer 0, and $111\ldots11$ represents -1. The two's-complement of a number is

$$2\text{comp}\,(k) = 1\text{comp}\,(\text{k}) + 1,$$

where 1comp (n) reverses the bits in n, and any remainder is discarded. It negates its argument. For instance,

$$\begin{aligned} 2\text{comp}(1) &= 1\text{comp}\left(\underbrace{00\ldots001}_{\text{n bits}}\right)+1 = \\ \underbrace{11\ldots110}_{\text{n bits}}+1 &= \underbrace{11\ldots111}_{\text{n bits}} = -1 \end{aligned}$$

The range of integers that can be represented is

$$-2^{n-1} \leq k \leq 2^{n-1}-1.$$

As an example, if an integer is stored using 32 bits, the range of integers is $2,147,483,648 \leq k \leq 2,147,483,647$.

To add, just perform binary addition using all n bits and discard any remainder. To subtract b from a, compute $a +$ 2comp (b). Addition or subtraction can overflow, meaning that the result cannot be represented using n bits. When this happens, the answer has the wrong sign.

Floating Point Format

A binary floating point number as described in this book has the form

$$\pm\left(0.d_1d_2\ldots d_p\right)\times b^n$$

with $d_1 \neq 0$, $d_i = 0, 1$, $-\text{e}_{\min} \leq n \leq \text{e}_{\max}$ is the exponent range, and p is the number of significant bits. Using this notation, the largest magnitude for a floating point number is $f_{\max} = 2^{\text{e}_{\max}}\left(1-2^{-p}\right)$, and smallest nonzero floating point number in magnitude is $f_{\min} = 2^{-(\text{e}_{\min}+1)}$.

Internally, the sign bit is the left-most bit, and 0 means nonnegative and 1 means negative. The exponent follows using e bits. To avoid having to represent negative exponents a bias of $2^{\text{e}-1}-1$ is added to the true exponent. For instance, if 8 bits are used for the exponent, the bias is 127. If the true exponent is -18, then the stored exponent is $-18+127 = 109 = 01101101_2$. The true exponent of zero is stored as $127 = 01111111$. The first binary digit $d_1 = 1$, and is the coefficient of $2^{-1} = \frac{1}{2}$. The remaining digits can be 0 or 1, and represent coefficients of $2^{-2}, 2^{-3}, \ldots$.

Since numbers like $\frac{1}{7} = 0.001001001001001001001001001001\ldots_2$ cannot be represented exactly using p digits, we round to p digits, and denote the stored number as fl (x). Doing this causes roundoff error, and this affects the accuracy of computations, sometimes causing serious problems.

Floating point numbers are granular, which means there are gaps between numbers. The gap is measured using the machine precision, eps, which is the distance between 1.0 and the next floating point number. In general, an interval from 2^k to 2^{k+1} has a gap between numbers of $2^k \times$ eps, and the gap relative to 2^k remains eps. If p binary digits are used, the value of eps is $\frac{1}{2}\times 2^{1-p}$.

IEEE single- and double-precision floating point arithmetic guarantees that

$$\text{fl}\,(x) = x\,(1+\epsilon)\,, \quad |\epsilon| \leq \text{eps}.$$

This is a fundamental formula when analyzing errors in floating point arithmetic.

Floating Point Arithmetic

Represent floating point addition of the true numbers a and b as $a \oplus b$. After computation, what we actually get is

$$a \oplus b = \text{fl}\,(\text{fl}\,(a)+\text{fl}\,(b))\,,$$

and normally roundoff error is present.

Measurement of Error

There are two ways to measure error, using absolute error or relative error:

$$\begin{aligned} \text{Absolute error} &= |\text{fl}\,(x)-x|\,, \\ \text{Relative error} &= \frac{|\text{fl}\,(x)-x|}{|x|}, \quad x \neq 0. \end{aligned}$$

Relative error is the most meaningful, as some examples in this chapter indicate. These types of error measurement apply to any calculation, not just measuring floating point error.

The analysis of roundoff error is complex, and we only do it for addition, presented in Theorem 8.1. Error bounds, as should be expected, involve eps.

Overflow and Underflow

Integer arithmetic can overflow, and the same is true for floating point arithmetic when the magnitude of a result exceeds the maximum allowable floating point number. In addition, underflow can occur, which means the magnitude of the result lies in the gap between 0 and the smallest floating point number.

Minimizing the Effects of Floating Point Error

Some computations are prone to floating point error, and should be replaced by an alternative. This chapter shows how to prevent overflow when computing

$$\|x\|_2 = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}.$$

Another type of error is caused by cancellation. A classic example is evaluation of the quadratic formula. We will encounter more situations where we must be careful how we perform a computation.

8.6 PROBLEMS

8.1 Find the base 10 representation for each number.
- **a.** 1101101101 (base 2)
- **b.** 33671 (base 8)
- **c.** 8FB2 (base 16)
- **d.** 221341 (base 5)

8.2 Write each unsigned decimal number in binary, octal (base 8), and hexadecimal (base 16).
- **a.** 45
- **b.** 167
- **c.** 273
- **d.** 32763

8.3 Perform unsigned binary addition. Do not discard the carry.

a.	b.	c.	d.
11011	11110101	001111	10101010
+	+	+	+
11101	10001001	011111	10101010

8.4 Perform unsigned binary subtraction. Use borrowing.

a.	b.	c.	d.
11110	11110101	001110	11101110
−	−	−	−
11101	10001001	001101	10101011

8.5 Using $b = 2$ and $p = 8$, find the two's-complement integer representation for
- **a.** 25
- **b.** 127
- **c.** -1
- **d.** -127
- **e.** -37
- **f.** -101

8.6 If $b = 2$ and $p = 6$, indicate which two's-complement sums will cause overflow and compute the result in binary.
- **a.** $-9 + 30$
- **b.** $28 + 5$
- **c.** $-25 - 7$
- **d.** $-32 + 23$

8.7 Assume a two's-complement integer representation using 10 binary bits.
- **a.** What is the range of integers that can be represented?
- **b.** For each addition, determine if overflow occurs.
  - **i.** $188 + 265$
  - **ii.** $490 + 25$
  - **iii.** $-400 + (-16)$
  - **iv.** $-450 + (-70)$

8.8 What is the range of two's complement numbers if $b = 8$ and $p = 15$?

8.9 A computer that uses two's-complement arithmetic for integers has a subtract machine instruction. However, when the instruction executes the CPU does not have to perform the subtraction by borrowing or even worry about signs. Why?

8.10 Another integer representation system is one's-complement. If an integer is stored using p bits, the positive integers are ordinary binary values with the left-most bit set to 0. For instance, if $p = 8$, then 65 is stored as

$$0 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1$$

Obtain the negative of a number by inverting bits; for instance, -65 is stored as

$$1 \quad 0 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 0$$

- **a.** Show how $1, -1, 25, 15, -21, 101, -120$ are stored in a one's-complement system with $b = 2, p = 8$.
- **b.** Show that there are two representations for 0 in a one's-complement system.
- **c.** Using base $b = 2$ and p digits, what is the range of one's-complement numbers?

8.11 Find the 32-bit single-precision representation for each number. Assume the format $\pm 0.d_1 d_2 \ldots d_{23} \times 2^e$, where $d_1 \neq 0$ unless the number is zero.
- **a.** 12.0625
- **b.** -18.1875
- **c.** 2.7

Note: In binary $0.7 = 0.10110011001100110011001100110011001100\ldots$

8.12 Assume you are performing decimal arithmetic with $p = 4$ significant digits. Using rounding, perform the following calculations:
- **a.** $26.8756 + 15.67883$
- **b.** $1.2567 * 14.77653$
- **c.** $12.98752 \times 10^3 * 23.47 \times 10^4$

8.13 Using four significant digits, compute the absolute and relative error for the following conversion to floating point form.
- **a.** $x_1 = 2.3566, \text{fl}(x_1) = 2.357$
- **b.** $x_2 = 7.1434, \text{fl}(x_2) = 7.143$

8.14 Verify the error bounds for addition and multiplication. Use $b = 10, p = 5$.
- **a.** $23.6643 + 45.6729 + 100.123$
- **b.** $8.25678 * 1.45729 * 5.35535$

8.15 Assume we are using IEEE double-precision arithmetic.
- **a.** What is the error bound in computing the sum of 20 positive floating point numbers?
- **b.** What is the error bound in computing the product of 20 floating point numbers?

8.16 If $b = 2, p = 12, \text{e}_{\min} = 5, \text{e}_{\max} = 8$, find $f_{\min}$ and $f_{\max}$.

8.17 By constructing a counterexample, verify that floating-point addition and multiplication, $\oplus$ and $\otimes$, do not obey the distributive law $a \otimes (b \oplus c) = a \otimes b + a \otimes c$.

8.18 Show that, unlike the operation $+$, the operation $\oplus$ is not associative. In other words $(x \oplus y) \oplus z \neq x \oplus (y \oplus z)$ in general.

8.19 Let $b = 10$ and $p = 10$. Compute $\text{fl}\left(A^2\right)$, where

$$A = \begin{bmatrix} 1 & 0 & 10^{-5} \\ 0 & 1 & 0 \\ 10^{-5} & 0 & 1 \end{bmatrix}.$$

Repeat your calculations with $p = 12$. Compare the results.

8.20 Assume you are using floating point arithmetic with $b = 10$, $p = 5$, and a maximum exponent of 8. Let x be the vector $x = \begin{bmatrix} 2500 \\ 6000 \\ 1000 \\ 8553 \end{bmatrix}$.

a. What is f_{max}?

b. Directly compute $\|x\| = \sqrt{2500^2 + 6000^2 + 1000^2 + 8553^2}$.

c. Compute $\|x\|$ using the method discussed in Section 8.4.1 that is designed to prevent overflow.

8.21 For parts (a) and (b), use $b = 10, p = 4$.

a. What is the truncation error when approximating cos (0.5) by the first two terms of its McLaurin series?

b. Answer part (a) for tan (0.5).

8.22 There are problems evaluating $f(x) = \sqrt{x-1} - \sqrt{x}$ under some conditions. Give an example. Propose a means of accurately computing $f(x)$.

8.23 Using $b = 10, p = 4$, verify the bound

$$\text{fl}(cA) = cA + E, \quad |E| \leq \text{eps}\,|cA|$$

for $c = 5.6797$ and

$$A = \begin{bmatrix} 2.34719 & -1.56219 & 5.89531 \\ -0.98431 & 23.764 & 102.35 \\ -77.543 & -0.87542 & 5.26743 \end{bmatrix}.$$

8.24 Find eps

a. for $b = 10, p = 8$.

b. for $b = 2, p = 128$.

8.25 Using floating point arithmetic, is it true that $a + b = a$ implies that $b = 0$? Justify your answer.

8.26 For parts (a) and (b), propose a method of computing each expression for small x. For part (c), propose a method for computing the expression for large x.

a. $\cos(x) - 1$

b. $\dfrac{\sin x - x}{x}$

c. $\dfrac{1}{x+1} - \dfrac{1}{x}$

8.27 Assume we have a floating point number system with base $b = 2$, p significant digits, minimum exponent e_{min} and maximum exponent e_{max}. A nonnegative number is either zero or of the form $0.1d_1d_2 \ldots d_{p-1} \times 2^e$. Develop a formula for the number of nonnegative floating point values.

8.6.1 MATLAB Problems

8.28 Taking a double value x and developing MATLAB code that precisely rounds it to m significant digits is somewhat difficult. Consider the function

```
function y = roundtom(x,m)
%ROUNDTOM round to m significant decimal digits
%
%   Input: floating point number x.
%          number of decimal digits desirede
%   Output: x rounded to m decimal digits

pos = floor(log10(abs(x)))-m+1;
y = round(x/10^pos)*10^pos;
```

Explain how the code works and test it with several double values. It often works perfectly but can leave some trailing nonzero digits. For instance

```
>> n = 23.567927;
>> roundtom(n,5)

ans =
  23.568000000000001
```

Explain why this occurs.

8.29 It is possible to write a function, say `fmex`, in the programming language C in such a way that the function can be called from MATLAB. The function `fmex` must written so it conforms with what is termed the MEX interface and must be compiled using the MATLAB command `mex` (see Ref. [20]). This interface provides access to the input and output parameters when the function is called from MATLAB. The book software distribution contains the C program outputdouble.c in the subdirectory outputdouble of the software distribution as well as compiled versions. Since machine code is system dependent, there are multiple versions. The following table lists the names of the available compiled code for Windows, OS X, and Linux systems.

Windows 64-bit	OS X 64-bit	Linux-64 bit
outputdouble.mexw64	outputdouble.mexmaci64	outputdouble.mexa64

If you are using a 32-bit system, execute "mex outputdouble.c", and MATLAB should generate 32-bit code. On any system, the calling format is

```
>> oututdouble(x)
```

where x is a double variable or a constant. It prints the 64 binary bits of x in IEEE double format, marking the location of the sign bit, the exponent, and the mantissa. The binary bits represent a floating point number of the form $\pm 1.d_1 d_2 d_3 \ldots d_{52} \times 2^e$. The leading 1 is hidden; in other words it is considered present but is not stored, giving 53 bits for the mantissa. The exponent uses excess 1023 format.

a. Find the binary representation for each number, determine the exponent in decimal, and the mantissa in binary.
- **i.** 33
- **ii.** -35
- **iii.** -101
- **iv.** 0.000677
- **v.** 0

b. The MATLAB named constants `realmax` and `realmin` are the largest and smallest double values. Using `outputdouble`, determine each number in binary and then determine what each number is in decimal.

8.30 The infinite series $\sum_{n=1}^{\infty} (-1)^{n+1}/n$ converges to $\ln(2)$. MATLAB performs computations using IEEE double-precision floating point arithmetic. Sum the first 100,000 terms of the series using MATLAB and determine the truncation error by accepting as correct the MATLAB value `log(2)`. Explain your result.

8.31 Type the command "`format hex`" in MATLAB to see the hexadecimal representation of any number. For 64-bit double-precision floating-point numbers, the first three hexadecimal digits correspond to the sign bit followed by the 11 exponent bits. The mantissa is represented by the next 13 hexadecimal digits. We have seen that as the numbers grow larger, the gaps between numbers grows as well.

a. Find 2^{75} in hexadecimal.

b. Investigate the gap between 2^{75} and the next floating-point number by finding the first number of the form $u = 2^{75} + 2^i$ that has a different hexadecimal representation.

c. What is the result of the following MATLAB statements `x = 2^75; y = x + 2^(i-1); x == y`. i is the power found in part (b). Explain the result.

d. For $b = 2, p = 3, e_{\min} = -2, e_{\max} = 3$, draw the distribution of floating-point numbers as in Figure 8.1.

8.32 The matrix exponential e^A is defined by the McLaurin series

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

This computation can be quite difficult, as this problem illustrates.

a. Enter the function

```
function E = matexp(A)
% MATEXP Taylor series for exp(A)

E = zeros(size(A));
F = eye(size(A));
k = 1;
while norm(E+F-E,1) > 0
   E = E + F;
   F = A*F/k;
   k = k+1;
end
```

that estimates e^A using the McLaurin series.

b. Apply `matexp` to the matrix

$$A = \begin{bmatrix} 99 & -100 \\ 137 & -138 \end{bmatrix}$$

to obtain the matrix `A_McLaurin`.

c. `A_McLaurin` is far from the correct result. Compute the true value of e^A using the MATLAB function `expm` as follows:

```
>> A_true = expm(A);
```

The function uses the Padé approximation [21].

d. Explain why the result in part (b) is so far from the correct result. Use MATLAB help by typing

```
>> showdemo expmdemo
```

Remark 8.8. The text by Laub [22, pp. 6-7] has a very interesting example dealing with the matrix exponential.

8.33 Write a MATLAB code segment that approximates eps. Start with epsest = 1.0 and halve epsest while 1.0 + epsest > 1.0. Run it, and compare the result with the MATLAB's `eps`.

8.34 This problem was developed from material on MATLAB Central (http://www.mathworks.com/matlabcentral/), and is due to Loren Shure.

a. Describe the action of the MATLAB function `fix`.

b. Enter and run the following code

c.
```
format short
for ind = 0:.1:1
    f = fix(10*ind)/10;
    if ind ~= f
        disp(ind - f);
    end
end
```

d. Explain why there is output when, theoretically, there should be none.

8.35 This problem experiments with cancellation errors when solving the quadratic equation (Section 8.4.2). The equation for the problem is

$$0.0001x^2 + 10000x - 0.0001 = 0.$$

a. Find the smallest root in magnitude of the polynomial using the quadratic equation.

b. Use the MATLAB function `roots` to determine the same root.

c. Which value is most correct?

8.36 Using double-precision arithmetic, determine the largest value of x for which e^x does not overflow. Compute $|e^x - \text{realmax}|$ and $(|e^x - \text{realmax}|)/\text{realmax}$. Is your result acceptable? Why?

8.37 Enter and run the following MATLAB code.

```
x = 0.0;
y = exp(-x);
k = 0;
while y ~= 0.0
    x = x + 0.01;
    y = exp(-x);
    k = k + 1;
    if mod(k,500) == 0
        fprintf('x = %g  y = %.16e\n', x, y);
    end
end
```

Execute the MATLAB command `realmin` that determines the smallest positive normalized floating-point number. Does your output appear to contradict the value of `realmin`? If it does, use the MATLAB documentation to explain the results. Hint: What is an unnormalized floating point number?