The McGraw-Hill Companies

Mc Graw Hill

NUMERICAL METHODS FOR ENGINEERS, FIFTH EDITION
International Edition 2006

10 09 08 07 06 05 04 03 02 01
20 09 08 07 06 05
CTF BJE

Library of Congress Control Number: 2005008475

When ordering this title, use ISBN 007-124429-8

Printed in Singapore

www.mhhe.com

CHAPTER 3

Approximations and Round-Off Errors

Because so many of the methods in this book are straightforward in description and application, it would be very tempting at this point for us to proceed directly to the main body of the text and teach you how to use these techniques. However, understanding the concept of error is so important to the effective use of numerical methods that we have chosen to devote the next two chapters to this topic.

The importance of error was introduced in our discussion of the falling parachutist in Chap. 1. Recall that we determined the velocity of a falling parachutist by both analytical and numerical methods. Although the numerical technique yielded estimates that were close to the exact analytical solution, there was a discrepancy, or *error,* because the numerical method involved an approximation. Actually, we were fortunate in that case because the availability of an analytical solution allowed us to compute the error exactly. For many applied engineering problems, we cannot obtain analytical solutions. Therefore, we cannot compute exactly the errors associated with our numerical methods. In these cases, we must settle for approximations or estimates of the errors.

Such errors are characteristic of most of the techniques described in this book. This statement might at first seem contrary to what one normally conceives of as sound engineering. Students and practicing engineers constantly strive to limit errors in their work. When taking examinations or doing homework problems, you are penalized, not rewarded, for your errors. In professional practice, errors can be costly and sometimes catastrophic. If a structure or device fails, lives can be lost.

Although perfection is a laudable goal, it is rarely, if ever, attained. For example, despite the fact that the model developed from Newton's second law is an excellent approximation, it would never in practice exactly predict the parachutist's fall. A variety of factors such as winds and slight variations in air resistance would result in deviations from the prediction. If these deviations are systematically high or low, then we might need to develop a new model. However, if they are randomly distributed and tightly grouped around the prediction, then the deviations might be considered negligible and the model deemed adequate. Numerical approximations also introduce similar discrepancies into the analysis. Again, the question is: How much error is present in our calculations and is it tolerable?

This chapter and the next cover basic topics related to the identification, quantification, and minimization of these errors. In this chapter, general information concerned with

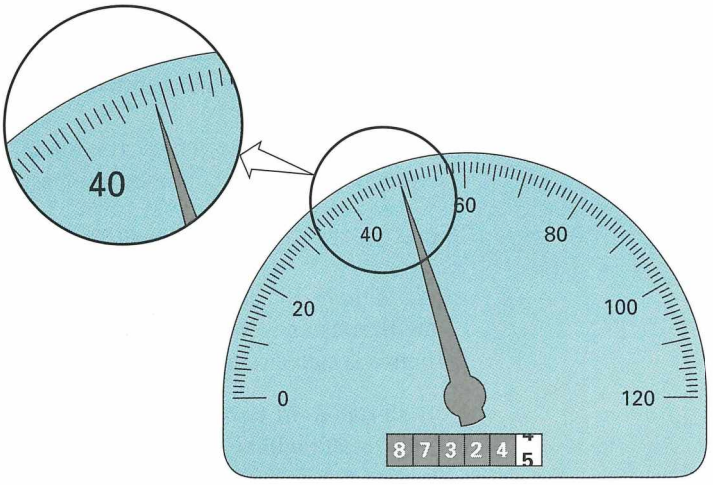
the quantification of error is reviewed in the first sections. This is followed by a section on one of the two major forms of numerical error: round-off error. *Round-off error* is due to the fact that computers can represent only quantities with a finite number of digits. Then Chap. 4 deals with the other major form: truncation error. *Truncation error* is the discrepancy introduced by the fact that numerical methods may employ approximations to represent exact mathematical operations and quantities. Finally, we briefly discuss errors not directly connected with the numerical methods themselves. These include blunders, formulation or model errors, and data uncertainty.

3.1 SIGNIFICANT FIGURES

This book deals extensively with approximations connected with the manipulation of numbers. Consequently, before discussing the errors associated with numerical methods, it is useful to review basic concepts related to approximate representation of the numbers themselves.

Whenever we employ a number in a computation, we must have assurance that it can be used with confidence. For example, Fig. 3.1 depicts a speedometer and odometer from an automobile. Visual inspection of the speedometer indicates that the car is traveling between 48 and 49 km/h. Because the indicator is higher than the midpoint between the markers on the gauge, we can say with assurance that the car is traveling at approximately 49 km/h. We have confidence in this result because two or more reasonable individuals reading this gauge would arrive at the same conclusion. However, let us say that we insist that the speed be estimated to one decimal place. For this case, one person might say 48.8,

FIGURE 3.1
An automobile speedometer and odometer illustrating the concept of a significant figure.



whereas another might say 48.9 km/h. Therefore, because of the limits of this instrument, only the first two digits can be used with confidence. Estimates of the third digit (or higher) must be viewed as approximations. It would be ludicrous to claim, on the basis of this speedometer, that the automobile is traveling at 48.8642138 km/h. In contrast, the odometer provides up to six certain digits. From Fig. 3.1, we can conclude that the car has traveled slightly less than 87,324.5 km during its lifetime. In this case, the seventh digit (and higher) is uncertain.

The concept of a significant figure, or digit, has been developed to formally designate the reliability of a numerical value. The *significant digits* of a number are those that can be used with confidence. They correspond to the number of certain digits plus one estimated digit. For example, the speedometer and the odometer in Fig. 3.1 yield readings of three and seven significant figures, respectively. For the speedometer, the two certain digits are 48. It is conventional to set the estimated digit at one-half of the smallest scale division on the measurement device. Thus the speedometer reading would consist of the three significant figures: 48.5. In a similar fashion, the odometer would yield a seven-significant-figure reading of 87,324.45.

Although it is usually a straightforward procedure to ascertain the significant figures of a number, some cases can lead to confusion. For example, zeros are not always significant figures because they may be necessary just to locate a decimal point. The numbers 0.00001845, 0.0001845, and 0.001845 all have four significant figures. Similarly, when trailing zeros are used in large numbers, it is not clear how many, if any, of the zeros are significant. For example, at face value the number 45,300 may have three, four, or five significant digits, depending on whether the zeros are known with confidence. Such uncertainty can be resolved by using scientific notation, where 4.53×10^4, 4.530×10^4, 4.5300×10^4 designate that the number is known to three, four, and five significant figures, respectively.

The concept of significant figures has two important implications for our study of numerical methods:

1. As introduced in the falling parachutist problem, numerical methods yield approximate results. We must, therefore, develop criteria to specify how confident we are in our approximate result. One way to do this is in terms of significant figures. For example, we might decide that our approximation is acceptable if it is correct to four significant figures.
2. Although quantities such as π, e, or $\sqrt{7}$ represent specific quantities, they cannot be expressed exactly by a limited number of digits. For example,

   $$\pi = 3.14159265358979323846264 3\ldots$$

   ad infinitum. Because computers retain only a finite number of significant figures, such numbers can never be represented exactly. The omission of the remaining significant figures is called round-off error.

Both round-off error and the use of significant figures to express our confidence in a numerical result will be explored in detail in subsequent sections. In addition, the concept of significant figures will have relevance to our definition of accuracy and precision in the next section.

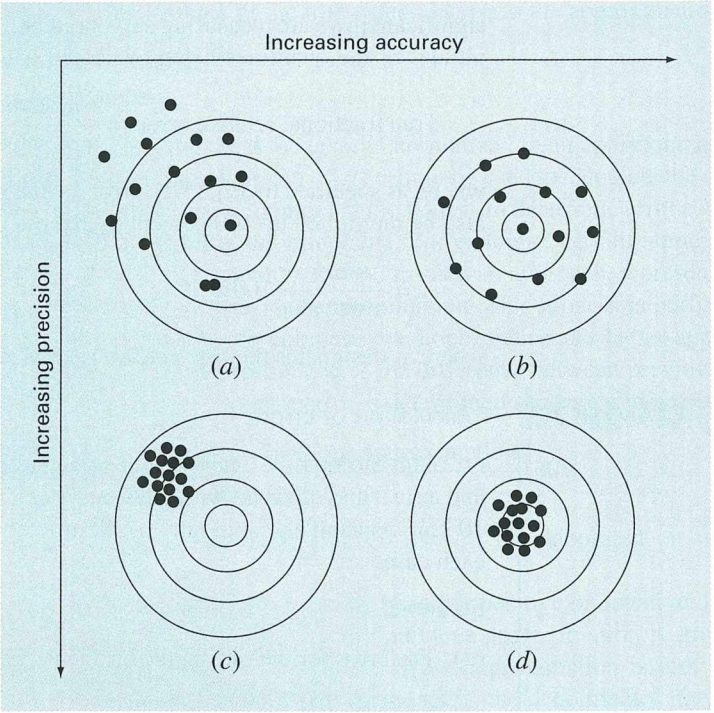
3.2 ACCURACY AND PRECISION

The errors associated with both calculations and measurements can be characterized with regard to their accuracy and precision. *Accuracy* refers to how closely a computed or measured value agrees with the true value. *Precision* refers to how closely individual computed or measured values agree with each other.

These concepts can be illustrated graphically using an analogy from target practice. The bullet holes on each target in Fig. 3.2 can be thought of as the predictions of a numerical technique, whereas the bull's-eye represents the truth. *Inaccuracy* (also called *bias*) is defined as systematic deviation from the truth. Thus, although the shots in Fig. 3.2*c* are more tightly grouped than those in Fig. 3.2*a*, the two cases are equally biased because they are both centered on the upper left quadrant of the target. *Imprecision* (also called *uncertainty*), on the other hand, refers to the magnitude of the scatter. Therefore, although Fig. 3.2*b* and *d* are equally accurate (that is, centered on the bull's-eye), the latter is more precise because the shots are tightly grouped.

Numerical methods should be sufficiently accurate or unbiased to meet the requirements of a particular engineering problem. They also should be precise enough for adequate

FIGURE 3.2
An example from marksmanship illustrating the concepts of accuracy and precision. (*a*) Inaccurate and imprecise; (*b*) accurate and imprecise; (*c*) inaccurate and precise; (*d*) accurate and precise.



engineering design. In this book, we will use the collective term *error* to represent both the inaccuracy and the imprecision of our predictions. With these concepts as background, we can now discuss the factors that contribute to the error of numerical computations.

3.3 ERROR DEFINITIONS

Numerical errors arise from the use of approximations to represent exact mathematical operations and quantities. These include *truncation errors,* which result when approximations are used to represent exact mathematical procedures, and *round-off errors,* which result when numbers having limited significant figures are used to represent exact numbers. For both types, the relationship between the exact, or true, result and the approximation can be formulated as

$$\text{True value} = \text{approximation} + \text{error} \tag{3.1}$$

By rearranging Eq. (3.1), we find that the numerical error is equal to the discrepancy between the truth and the approximation, as in

$$E_t = \text{true value} - \text{approximation} \tag{3.2}$$

where E_t is used to designate the exact value of the error. The subscript t is included to designate that this is the "true" error. This is in contrast to other cases, as described shortly, where an "approximate" estimate of the error must be employed.

A shortcoming of this definition is that it takes no account of the order of magnitude of the value under examination. For example, an error of a centimeter is much more significant if we are measuring a rivet rather than a bridge. One way to account for the magnitudes of the quantities being evaluated is to normalize the error to the true value, as in

$$\text{True fractional relative error} = \frac{\text{true error}}{\text{true value}}$$

where, as specified by Eq. (3.2), error = true value − approximation. The relative error can also be multiplied by 100 percent to express it as

$$\varepsilon_t = \frac{\text{true error}}{\text{true value}} 100\% \tag{3.3}$$

where ε_t designates the true percent relative error.

EXAMPLE 3.1 Calculation of Errors

Problem Statement. Suppose that you have the task of measuring the lengths of a bridge and a rivet and come up with 9999 and 9 cm, respectively. If the true values are 10,000 and 10 cm, respectively, compute (*a*) the true error and (*b*) the true percent relative error for each case.

Solution.

(a) The error for measuring the bridge is [Eq. (3.2)]

$$E_t = 10{,}000 - 9999 = 1 \text{ cm}$$

and for the rivet it is

$$E_t = 10 - 9 = 1 \text{ cm}$$

(b) The percent relative error for the bridge is [Eq. (3.3)]

$$\varepsilon_t = \frac{1}{10{,}000} 100\% = 0.01\%$$

and for the rivet it is

$$\varepsilon_t = \frac{1}{10} 100\% = 10\%$$

Thus, although both measurements have an error of 1 cm, the relative error for the rivet is much greater. We would conclude that we have done an adequate job of measuring the bridge, whereas our estimate for the rivet leaves something to be desired.

Notice that for Eqs. (3.2) and (3.3), E and ε are subscripted with a t to signify that the error is normalized to the true value. In Example 3.1, we were provided with this value. However, in actual situations such information is rarely available. For numerical methods, the true value will be known only when we deal with functions that can be solved analytically. Such will typically be the case when we investigate the theoretical behavior of a particular technique for simple systems. However, in real-world applications, we will obviously not know the true answer a priori. For these situations, an alternative is to normalize the error using the best available estimate of the true value, that is, to the approximation itself, as in

$$\varepsilon_a = \frac{\text{approximate error}}{\text{approximation}} 100\% \tag{3.4}$$

where the subscript a signifies that the error is normalized to an approximate value. Note also that for real-world applications, Eq. (3.2) cannot be used to calculate the error term for Eq. (3.4). One of the challenges of numerical methods is to determine error estimates in the absence of knowledge regarding the true value. For example, certain numerical methods use an *iterative approach* to compute answers. In such an approach, a present approximation is made on the basis of a previous approximation. This process is performed repeatedly, or iteratively, to successively compute (we hope) better and better approximations. For such cases, the error is often estimated as the difference between previous and current approximations. Thus, percent relative error is determined according to

$$\varepsilon_a = \frac{\text{current approximation} - \text{previous approximation}}{\text{current approximation}} 100\% \tag{3.5}$$

This and other approaches for expressing errors will be elaborated on in subsequent chapters.

The signs of Eqs. (3.2) through (3.5) may be either positive or negative. If the approximation is greater than the true value (or the previous approximation is greater than the current approximation), the error is negative; if the approximation is less than the true value, the error is positive. Also, for Eqs. (3.3) to (3.5), the denominator may be less than

zero, which can also lead to a negative error. Often, when performing computations, we may not be concerned with the sign of the error, but we are interested in whether the percent absolute value is lower than a prespecified percent tolerance ε_s. Therefore, it is often useful to employ the absolute value of Eqs. (3.2) through (3.5). For such cases, the computation is repeated until

$$|\varepsilon_a| < \varepsilon_s \tag{3.6}$$

If this relationship holds, our result is assumed to be within the prespecified acceptable level ε_s. Note that for the remainder of this text, we will almost exclusively employ absolute values when we use relative errors.

It is also convenient to relate these errors to the number of significant figures in the approximation. It can be shown (Scarborough, 1966) that if the following criterion is met, we can be assured that the result is correct to *at least* n significant figures.

$$\varepsilon_s = (0.5 \times 10^{2-n})\% \tag{3.7}$$

EXAMPLE 3.2 Error Estimates for Iterative Methods

Problem Statement. In mathematics, functions can often be represented by infinite series. For example, the exponential function can be computed using

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!} \tag{E3.2.1}$$

Thus, as more terms are added in sequence, the approximation becomes a better and better estimate of the true value of e^x. Equation (E3.2.1) is called a *Maclaurin series expansion.*

Starting with the simplest version, $e^x = 1$, add terms one at a time to estimate $e^{0.5}$. After each new term is added, compute the true and approximate percent relative errors with Eqs. (3.3) and (3.5), respectively. Note that the true value is $e^{0.5} = 1.648721\ldots$. Add terms until the absolute value of the approximate error estimate ε_a falls below a prespecified error criterion ε_s conforming to three significant figures.

Solution. First, Eq. (3.7) can be employed to determine the error criterion that ensures a result is correct to at least three significant figures:

$$\varepsilon_s = (0.5 \times 10^{2-3})\% = 0.05\%$$

Thus, we will add terms to the series until ε_a falls below this level.

The first estimate is simply equal to Eq. (E3.2.1) with a single term. Thus, the first estimate is equal to 1. The second estimate is then generated by adding the second term, as in

$$e^x = 1 + x$$

or for $x = 0.5$,

$$e^{0.5} = 1 + 0.5 = 1.5$$

This represents a true percent relative error of [Eq. (3.3)]

$$\varepsilon_t = \frac{1.648721 - 1.5}{1.648721} 100\% = 9.02\%$$

Equation (3.5) can be used to determine an approximate estimate of the error, as in

$$\varepsilon_a = \frac{1.5 - 1}{1.5} 100\% = 33.3\%$$

Because ε_a is not less than the required value of ε_s,we would continue the computation by adding another term, $x^2/2!$, and repeating the error calculations. The process is continued until $\varepsilon_a < \varepsilon_s$. The entire computation can be summarized as

Terms	Result	ε_t (%)	ε_a (%)
1	1	39.3	
2	1.5	9.02	33.3
3	1.625	1.44	7.69
4	1.645833333	0.175	1.27
5	1.648437500	0.0172	0.158
6	1.648697917	0.00142	0.0158

Thus, after six terms are included, the approximate error falls below $\varepsilon_s = 0.05\%$ and the computation is terminated. However, notice that, rather than three significant figures, the result is accurate to five! This is because, for this case, both Eqs. (3.5) and (3.7) are conservative. That is, they ensure that the result is at least as good as they specify. Although, as discussed in Chap. 6, this is not always the case for Eq. (3.5), it is true most of the time.

With the preceding definitions as background, we can now proceed to the two types of error connected directly with numerical methods: round-off errors and truncation errors.

3.4 ROUND-OFF ERRORS

As mentioned previously, round-off errors originate from the fact that computers retain only a fixed number of significant figures during a calculation. Numbers such as π, e, or $\sqrt{7}$ cannot be expressed by a fixed number of significant figures. Therefore, they cannot be represented exactly by the computer. In addition, because computers use a base-2 representation, they cannot precisely represent certain exact base-10 numbers. The discrepancy introduced by this omission of significant figures is called *round-off error.*

3.4.1 Computer Representation of Numbers

Numerical round-off errors are directly related to the manner in which numbers are stored in a computer. The fundamental unit whereby information is represented is called a *word.* This is an entity that consists of a string of *b*inary dig*its,* or *bits.* Numbers are typically stored in one or more words. To understand how this is accomplished, we must first review some material related to number systems.

Number Systems. A *number system* is merely a convention for representing quantities. Because we have 10 fingers and 10 toes, the number system that we are most familiar with is the *decimal,* or *base*-10, number system. A base is the number used as the reference for

constructing the system. The base-10 system uses the 10 digits—0, 1, 2, 3, 4, 5, 6, 7, 8, 9—to represent numbers. By themselves, these digits are satisfactory for counting from 0 to 9.

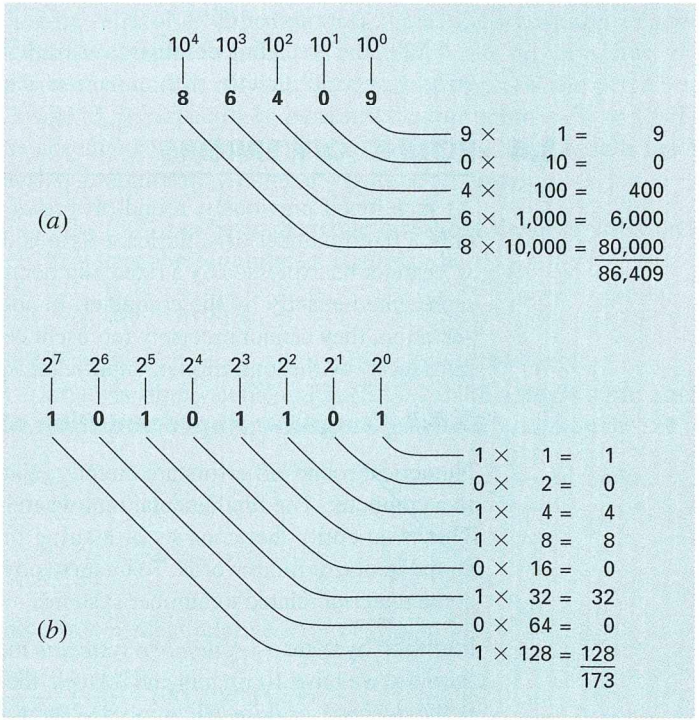
For larger quantities, combinations of these basic digits are used, with the position or *place value* specifying the magnitude. The right-most digit in a whole number represents a number from 0 to 9. The second digit from the right represents a multiple of 10. The third digit from the right represents a multiple of 100 and so on. For example, if we have the number 86,409 then we have eight groups of 10,000, six groups of 1000, four groups of 100, zero groups of 10, and nine more units, or

$$(8 \times 10^4) + (6 \times 10^3) + (4 \times 10^2) + (0 \times 10^1) + (9 \times 10^0) = 86{,}409$$

Figure 3.3*a* provides a visual representation of how a number is formulated in the base-10 system. This type of representation is called *positional notation.*

Because the decimal system is so familiar, it is not commonly realized that there are alternatives. For example, if human beings happened to have had eight fingers and eight toes, we would undoubtedly have developed an *octal,* or *base*-8, representation. In the same sense, our friend the computer is like a two-fingered animal who is limited to two states—either 0 or 1. This relates to the fact that the primary logic units of digital computers

FIGURE 3.3
How the (*a*) decimal (base 10) and the (*b*) binary (base 2) systems work. In (*b*), the binary number 10101101 is equivalent to the decimal number 173.



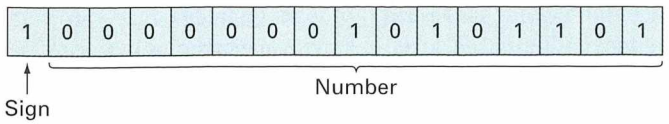


FIGURE 3.4
The representation of the decimal integer −173 on a 16-bit computer using the signed magnitude method.

are on/off electronic components. Hence, numbers on the computer are represented with a *binary,* or *base*-2, system. Just as with the decimal system, quantities can be represented using positional notation. For example, the binary number 11 is equivalent to $(1 \times 2^1) + (1 \times 2^0) = 2 + 1 = 3$ in the decimal system. Figure 3.3*b* illustrates a more complicated example.

Integer Representation. Now that we have reviewed how base-10 numbers can be represented in binary form, it is simple to conceive of how integers are represented on a computer. The most straightforward approach, called the *signed magnitude method,* employs the first bit of a word to indicate the sign, with a 0 for positive and a 1 for negative. The remaining bits are used to store the number. For example, the integer value of −173 would be stored on a 16-bit computer, as in Fig. 3.4.

EXAMPLE 3.3 Range of Integers

Problem Statement. Determine the range of integers in base-10 that can be represented on a 16-bit computer.

Solution. Of the 16 bits, the first bit holds the sign. The remaining 15 bits can hold binary numbers from 0 to 111111111111111. The upper limit can be converted to a decimal integer, as in

$$(1 \times 2^{14}) + (1 \times 2^{13}) + \cdots + (1 \times 2^1) + (1 \times 2^0)$$

which equals 32,767 (note that this expression can be simply evaluated as $2^{15} - 1$). Thus, a 16-bit computer word can store decimal integers ranging from −32,767 to 32,767. In addition, because zero is already defined as 0000000000000000, it is redundant to use the number 1000000000000000 to define a "minus zero." Therefore, it is usually employed to represent an additional negative number: −32,768, and the range is from −32,768 to 32,767.

Note that the signed magnitude method described above is not used to represent integers on conventional computers. A preferred approach called the *2's complement* technique directly incorporates the sign into the number's magnitude rather than providing a separate bit to represent plus or minus (see Chapra and Canale 1994). However, Example 3.3 still serves to illustrate how all digital computers are limited in their capability to represent integers. That is, numbers above or below the range cannot be represented. A more serious

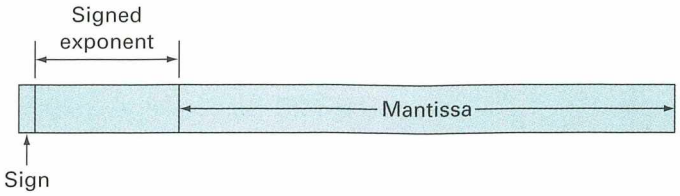


FIGURE 3.5
The manner in which a floating-point number is stored in a word.

limitation is encountered in the storage and manipulation of fractional quantities as described next.

Floating-Point Representation. Fractional quantities are typically represented in computers using floating-point form. In this approach, the number is expressed as a fractional part, called a *mantissa* or *significand,* and an integer part, called an *exponent* or *characteristic,* as in

$$m \cdot b^e$$

where m = the mantissa, b = the base of the number system being used, and e = the exponent. For instance, the number 156.78 could be represented as 0.15678×10^3 in a floating-point base-10 system.

Figure 3.5 shows one way that a floating-point number could be stored in a word. The first bit is reserved for the sign, the next series of bits for the signed exponent, and the last bits for the mantissa.

Note that the mantissa is usually *normalized* if it has leading zero digits. For example, suppose the quantity $1/34 = 0.029411765\ldots$ was stored in a floating-point base-10 system that allowed only four decimal places to be stored. Thus, $1/34$ would be stored as

$$0.0294 \times 10^0$$

However, in the process of doing this, the inclusion of the useless zero to the right of the decimal forces us to drop the digit 1 in the fifth decimal place. The number can be normalized to remove the leading zero by multiplying the mantissa by 10 and lowering the exponent by 1 to give

$$0.2941 \times 10^{-1}$$

Thus, we retain an additional significant figure when the number is stored.

The consequence of normalization is that the absolute value of m is limited. That is,

$$\frac{1}{b} \leq m < 1 \tag{3.8}$$

where b = the base. For example, for a base-10 system, m would range between 0.1 and 1, and for a base-2 system, between 0.5 and 1.

Floating-point representation allows both fractions and very large numbers to be expressed on the computer. However, it has some disadvantages. For example, floating-point

numbers take up more room and take longer to process than integer numbers. More significantly, however, their use introduces a source of error because the mantissa holds only a finite number of significant figures. Thus, a round-off error is introduced.

EXAMPLE 3.4 Hypothetical Set of Floating-Point Numbers

Problem Statement. Create a hypothetical floating-point number set for a machine that stores information using 7-bit words. Employ the first bit for the sign of the number, the next three for the sign and the magnitude of the exponent, and the last three for the magnitude of the mantissa (Fig. 3.6).

Solution. The smallest possible positive number is depicted in Fig. 3.6. The initial 0 indicates that the quantity is positive. The 1 in the second place designates that the exponent has a negative sign. The 1's in the third and fourth places give a maximum value to the exponent of

$$1 \times 2^1 + 1 \times 2^0 = 3$$

Therefore, the exponent will be -3. Finally, the mantissa is specified by the 100 in the last three places, which conforms to

$$1 \times 2^{-1} + 0 \times 2^{-2} + 0 \times 2^{-3} = 0.5$$

Although a smaller mantissa is possible (e.g., 000, 001, 010, 011), the value of 100 is used because of the limit imposed by normalization [Eq. (3.8)]. Thus, the smallest possible positive number for this system is $+0.5 \times 2^{-3}$, which is equal to 0.0625 in the base-10 system. The next highest numbers are developed by increasing the mantissa, as in

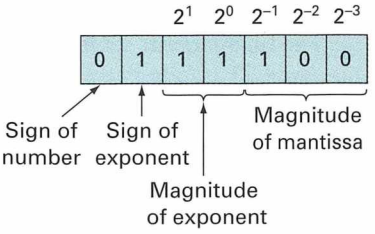
$$0111101 = (1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}) \times 2^{-3} = (0.078125)_{10}$$
$$0111110 = (1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3}) \times 2^{-3} = (0.093750)_{10}$$
$$0111111 = (1 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}) \times 2^{-3} = (0.109375)_{10}$$

Notice that the base-10 equivalents are spaced evenly with an interval of 0.015625.

At this point, to continue increasing, we must decrease the exponent to 10, which gives a value of

$$1 \times 2^1 + 0 \times 2^0 = 2$$

FIGURE 3.6
The smallest possible positive floating-point number from Example 3.4.



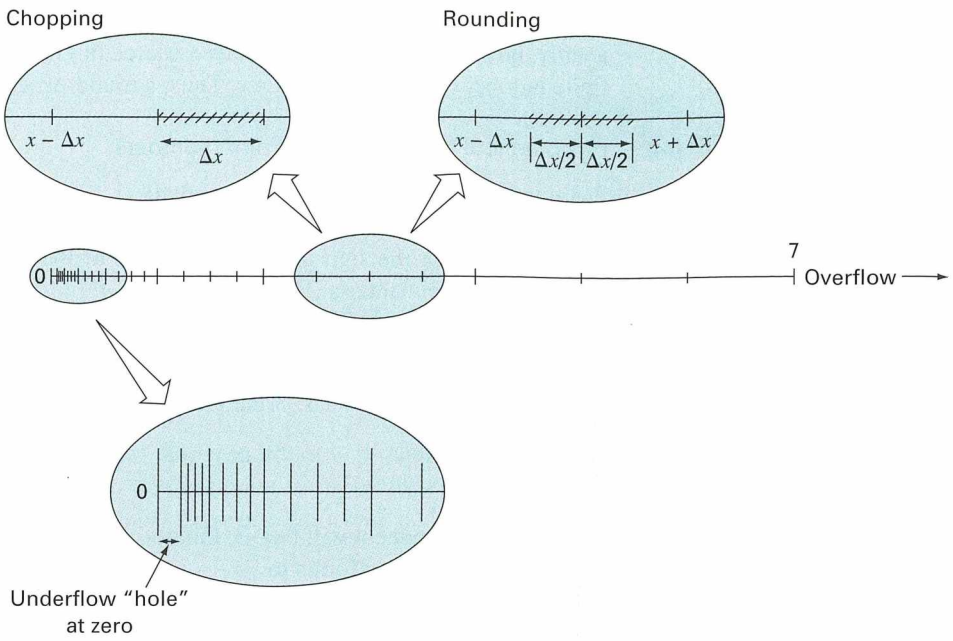


FIGURE 3.7
The hypothetical number system developed in Example 3.4. Each value is indicated by a tick mark. Only the positive numbers are shown. An identical set would also extend in the negative direction.

The mantissa is decreased back to its smallest value of 100. Therefore, the next number is

$$0110100 = (1 \times 2^{-1} + 0 \times 2^{-2} + 0 \times 2^{-3}) \times 2^{-2} = (0.125000)_{10}$$

This still represents a gap of $0.125000 - 0.109375 = 0.015625$. However, now when higher numbers are generated by increasing the mantissa, the gap is lengthened to 0.03125,

$$0110101 = (1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}) \times 2^{-2} = (0.156250)_{10}$$
$$0110110 = (1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3}) \times 2^{-2} = (0.187500)_{10}$$
$$0110111 = (1 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}) \times 2^{-2} = (0.218750)_{10}$$

This pattern is repeated as each larger quantity is formulated until a maximum number is reached,

$$0011111 = (1 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}) \times 2^{3} = (7)_{10}$$

The final number set is depicted graphically in Fig. 3.7.

Figure 3.7 manifests several aspects of floating-point representation that have significance regarding computer round-off errors:

1. *There Is a Limited Range of Quantities That May Be Represented.* Just as for the integer case, there are large positive and negative numbers that cannot be represented.

Attempts to employ numbers outside the acceptable range will result in what is called an *overflow error*. However, in addition to large quantities, the floating-point representation has the added limitation that very small numbers cannot be represented. This is illustrated by the *underflow* "hole" between zero and the first positive number in Fig. 3.7. It should be noted that this hole is enlarged because of the normalization constraint of Eq. (3.8).

2. *There Are Only a Finite Number of Quantities That Can Be Represented within the Range.* Thus, the degree of precision is limited. Obviously, irrational numbers cannot be represented exactly. Furthermore, rational numbers that do not exactly match one of the values in the set also cannot be represented precisely. The errors introduced by approximating both these cases are referred to as *quantizing* errors. The actual approximation is accomplished in either of two ways: chopping or rounding. For example, suppose that the value of $\pi = 3.14159265358\ldots$ is to be stored on a base-10 number system carrying seven significant figures. One method of approximation would be to merely omit, or "chop off," the eighth and higher terms, as in $\pi = 3.141592$, with the introduction of an associated error of [Eq. (3.2)]

$$E_t = 0.00000065\ldots$$

This technique of retaining only the significant terms was originally dubbed "truncation" in computer jargon. We prefer to call it *chopping* to distinguish it from the truncation errors discussed in Chap. 4. Note that for the base-2 number system in Fig. 3.7, chopping means that any quantity falling within an interval of length Δx will be stored as the quantity at the lower end of the interval. Thus, the upper error bound for chopping is Δx. Additionally, a bias is introduced because all errors are positive. The shortcomings of chopping are attributable to the fact that the higher terms in the complete decimal representation have no impact on the shortened version. For instance, in our example of π, the first discarded digit is 6. Thus, the last retained digit should be rounded up to yield 3.141593. Such *rounding* reduces the error to

$$E_t = -0.00000035\ldots$$

Consequently, rounding yields a lower absolute error than chopping. Note that for the base-2 number system in Fig. 3.7, rounding means that any quantity falling within an interval of length Δx will be represented as the nearest allowable number. Thus, the upper error bound for rounding is $\Delta x/2$. Additionally, no bias is introduced because some errors are positive and some are negative. Some computers employ rounding. However, this adds to the computational overhead, and, consequently, many machines use simple chopping. This approach is justified under the supposition that the number of significant figures is large enough that resulting round-off error is usually negligible.

3. *The Interval between Numbers, Δx, Increases as the Numbers Grow in Magnitude.* It is this characteristic, of course, that allows floating-point representation to preserve significant digits. However, it also means that quantizing errors will be proportional to the magnitude of the number being represented. For normalized floating-point numbers, this proportionality can be expressed, for cases where chopping is employed, as

$$\frac{|\Delta x|}{|x|} \leq \mathscr{E} \tag{3.9}$$

and, for cases where rounding is employed, as

$$\frac{|\Delta x|}{|x|} \leq \frac{\mathscr{E}}{2} \tag{3.10}$$

where $\mathscr{E}$ is referred to as the *machine epsilon,* which can be computed as

$$\mathscr{E} = b^{1-t} \tag{3.11}$$

where b is the number base and t is the number of significant digits in the mantissa. Notice that the inequalities in Eqs. (3.9) and (3.10) signify that these are error bounds. That is, they specify the worst cases.

EXAMPLE 3.5 Machine Epsilon

Problem Statement. Determine the machine epsilon and verify its effectiveness in characterizing the errors of the number system from Example 3.4. Assume that chopping is used.

Solution. The hypothetical floating-point system from Example 3.4 employed values of the base $b = 2$, and the number of mantissa bits $t = 3$. Therefore, the machine epsilon would be [Eq. (3.11)]

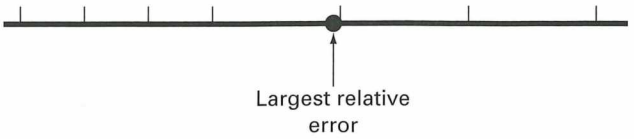
$$\mathscr{E} = 2^{1-3} = 0.25$$

Consequently, the relative quantizing error should be bounded by 0.25 for chopping. The largest relative errors should occur for those quantities that fall just below the upper bound of the first interval between successive equispaced numbers (Fig. 3.8). Those numbers falling in the succeeding higher intervals would have the same value of Δx but a greater value of x and, hence, would have a lower relative error. An example of a maximum error would be a value falling just below the upper bound of the interval between $(0.125000)_{10}$ and $(0.156250)_{10}$. For this case, the error would be less than

$$\frac{0.03125}{0.125000} = 0.25$$

Thus, the error is as predicted by Eq. (3.9).

FIGURE 3.8
The largest quantizing error will occur for those values falling just below the upper bound of the first of a series of equispaced intervals.



```
epsilon = 1
DO
  IF (epsilon+1 ≤ 1) EXIT
  epsilon = epsilon/2
END DO
epsilon = 2 × epsilon
```

FIGURE 3.9
Pseudocode to determine machine epsilon for a binary computer.

The magnitude dependence of quantizing errors has a number of practical applications in numerical methods. Most of these relate to the commonly employed operation of testing whether two numbers are equal. This occurs when testing convergence of quantities as well as in the stopping mechanism for iterative processes (recall Example 3.2). For these cases, it should be clear that, rather than test whether the two quantities are equal, it is advisable to test whether their difference is less than an acceptably small tolerance. Further, it should also be evident that normalized rather than absolute difference should be compared, particularly when dealing with numbers of large magnitude. In addition, the machine epsilon can be employed in formulating stopping or convergence criteria. This ensures that programs are portable—that is, they are not dependent on the computer on which they are implemented. Figure 3.9 lists pseudocode to automatically determine the machine epsilon of a binary computer.

Extended Precision. It should be noted at this point that, although round-off errors can be important in contexts such as testing convergence, the number of significant digits carried on most computers allows most engineering computations to be performed with more than acceptable precision. For example, the hypothetical number system in Fig. 3.7 is a gross exaggeration that was employed for illustrative purposes. Commercial computers use much larger words and, consequently, allow numbers to be expressed with more than adequate precision. For example, computers that use IEEE format allow 24 bits to be used for the mantissa, which translates into about seven significant base-10 digits of precision[1] with a range of about 10^{-38} to 10^{39}.

With this acknowledged, there are still cases where round-off error becomes critical. For this reason most computers allow the specification of extended precision. The most common of these is double precision, in which the number of words used to store floating-point numbers is doubled. It provides about 15 to 16 decimal digits of precision and a range of approximately 10^{-308} to 10^{308}.

In many cases, the use of double-precision quantities can greatly mitigate the effect of round-off errors. However, a price is paid for such remedies in that they also require more memory and execution time. The difference in execution time for a small calculation might seem insignificant. However, as your programs become larger and more complicated, the added execution time could become considerable and have a negative impact on your effectiveness as a problem solver. Therefore, extended precision should not be used frivolously. Rather, it should be selectively employed where it will yield the maximum benefit at the least cost in terms of execution time. In the following sections, we will look closer at how round-off errors affect computations, and in so doing provide a foundation of understanding to guide your use of the double-precision capability.

Before proceeding, it should be noted that some of the commonly used software packages (for example, Excel, Mathcad) routinely use double precision to represent numerical quantities. Thus, the developers of these packages decided that mitigating round-off errors would take precedence over any loss of speed incurred by using extended precision. Others, like MATLAB software, allow you to use extended precision, if you desire.

[1]Note that only 23 bits are actually used to store the mantissa. However, because of normalization, the first bit of the mantissa is always 1 and is, therefore, not stored. Thus, this first bit together with the 23 stored bits gives the 24 total bits of precision for the mantissa.

3.4.2 Arithmetic Manipulations of Computer Numbers

Aside from the limitations of a computer's number system, the actual arithmetic manipulations involving these numbers can also result in round-off error. In the following section, we will first illustrate how common arithmetic operations affect round-off errors. Then we will investigate a number of particular manipulations that are especially prone to round-off errors.

Common Arithmetic Operations. Because of their familiarity, normalized base-10 numbers will be employed to illustrate the effect of round-off errors on simple addition, subtraction, multiplication, and division. Other number bases would behave in a similar fashion. To simplify the discussion, we will employ a hypothetical decimal computer with a 4-digit mantissa and a 1-digit exponent. In addition, chopping is used. Rounding would lead to similar though less dramatic errors.

When two floating-point numbers are added, the mantissa of the number with the smaller exponent is modified so that the exponents are the same. This has the effect of aligning the decimal points. For example, suppose we want to add $0.1557 \cdot 10^1 + 0.4381 \cdot 10^{-1}$. The decimal of the mantissa of the second number is shifted to the left a number of places equal to the difference of the exponents $[1 - (-1) = 2]$, as in

$$0.4381 \cdot 10^{-1} \rightarrow 0.004381 \cdot 10^1$$

Now the numbers can be added,

$$\begin{array}{l} 0.1557 \quad \cdot 10^1 \\ \underline{0.004381 \cdot 10^1} \\ 0.160081 \cdot 10^1 \end{array}$$

and the result chopped to $0.1600 \cdot 10^1$. Notice how the last two digits of the second number that were shifted to the right have essentially been lost from the computation.

Subtraction is performed identically to addition except that the sign of the subtrahend is reversed. For example, suppose that we are subtracting 26.86 from 36.41. That is,

$$\begin{array}{r} 0.3641 \cdot 10^2 \\ \underline{-\ 0.2686 \cdot 10^2} \\ 0.0955 \cdot 10^2 \end{array}$$

For this case the result is not normalized, and so we must shift the decimal one place to the right to give $0.9550 \cdot 10^1 = 9.550$. Notice that the zero added to the end of the mantissa is not significant but is merely appended to fill the empty space created by the shift. Even more dramatic results would be obtained when the numbers are very close, as in

$$\begin{array}{r} 0.7642 \cdot 10^3 \\ \underline{-\ 0.7641 \cdot 10^3} \\ 0.0001 \cdot 10^3 \end{array}$$

which would be converted to $0.1000 \cdot 10^0 = 0.1000$. Thus, for this case, three nonsignificant zeros are appended. This introduces a substantial computational error because subsequent manipulations would act as if these zeros were significant. As we will see in a later section, the loss of significance during the subtraction of nearly equal numbers is among the greatest source of round-off error in numerical methods.

Multiplication and division are somewhat more straightforward than addition or subtraction. The exponents are added and the mantissas multiplied. Because multiplication of two n-digit mantissas will yield a $2n$-digit result, most computers hold intermediate results

in a double-length register. For example,

$$0.1363 \cdot 10^3 \times 0.6423 \cdot 10^{-1} = 0.08754549 \cdot 10^2$$

If, as in this case, a leading zero is introduced, the result is normalized,

$$0.08754549 \cdot 10^2 \rightarrow 0.8754549 \cdot 10^1$$

and chopped to give

$$0.8754 \cdot 10^1$$

Division is performed in a similar manner, but the mantissas are divided and the exponents are subtracted. Then the results are normalized and chopped.

Large Computations. Certain methods require extremely large numbers of arithmetic manipulations to arrive at their final results. In addition, these computations are often interdependent. That is, the later calculations are dependent on the results of earlier ones. Consequently, even though an individual round-off error could be small, the cumulative effect over the course of a large computation can be significant.

EXAMPLE 3.6 Large Numbers of Interdependent Computations

Problem Statement. Investigate the effect of round-off error on large numbers of interdependent computations. Develop a program to sum a number 100,000 times. Sum the number 1 in single precision, and 0.00001 in single and double precision.

Solution. Figure 3.10 shows a Fortran 90 program that performs the summation. Whereas the single-precision summation of 1 yields the expected result, the single-precision

FIGURE 3.10
Fortran 90 program to sum a number 10^5 times. The case sums the number 1 in single precision and the number 10^{-5} in single and double precision.

```
PROGRAM fig0310
IMPLICIT none
INTEGER::i
REAL::sum1, sum2, x1, x2
DOUBLE PRECISION::sum3, x3
sum1=0.
sum2=0.
sum3=0.
x1=1.
x2=1.e-5
x3=1.d-5
DO i=1,100000
  sum1=sum1+x1
  sum2=sum2+x2
  sum3=sum3+x3
END DO
PRINT *, sum1
PRINT *, sum2
PRINT *, sum3
END
output:
100000.000000
       1.000990
  9.999999999980838E-001
```

summation of 0.00001 yields a large discrepancy. This error is reduced significantly when 0.00001 is summed in double precision.

Quantizing errors are the source of the discrepancies. Because the integer 1 can be represented exactly within the computer, it can be summed exactly. In contrast, 0.00001 cannot be represented exactly and is quantized by a value that is slightly different from its true value. Whereas this very slight discrepancy would be negligible for a small computation, it accumulates after repeated summations. The problem still occurs in double precision but is greatly mitigated because the quantizing error is much smaller.

Note that the type of error illustrated by the previous example is somewhat atypical in that all the errors in the repeated operation are of the same sign. In most cases the errors of a long computation alternate sign in a random fashion and, thus, often cancel out. However, there are also instances where such errors do not cancel but, in fact, lead to a spurious final result. The following sections are intended to provide insight into ways in which this may occur.

Adding a Large and a Small Number. Suppose we add a small number, 0.0010, to a large number, 4000, using a hypothetical computer with the 4-digit mantissa and the 1-digit exponent. We modify the smaller number so that its exponent matches the larger,

$$\begin{array}{l} 0.4000 \quad \cdot 10^4 \\ \underline{0.0000001 \cdot 10^4} \\ 0.4000001 \cdot 10^4 \end{array}$$

which is chopped to $0.4000 \cdot 10^4$. Thus, we might as well have not performed the addition!

This type of error can occur in the computation of an infinite series. The initial terms in such series are often relatively large in comparison with the later terms. Thus, after a few terms have been added, we are in the situation of adding a small quantity to a large quantity.

One way to mitigate this type of error is to sum the series in reverse order—that is, in ascending rather than descending order. In this way, each new term will be of comparable magnitude to the accumulated sum (see Prob. 3.4).

Subtractive Cancellation. This term refers to the round-off induced when subtracting two nearly equal floating-point numbers.

One common instance where this can occur involves finding the roots of a quadratic equation or parabola with the quadratic formula,

$$\begin{matrix} x_1 \\ x_2 \end{matrix} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{3.12}$$

For cases where $b^2 \gg 4ac$, the difference in the numerator can be very small. In such cases, double precision can mitigate the problem. In addition, an alternative formulation can be used to minimize subtractive cancellation,

$$\begin{matrix} x_1 \\ x_2 \end{matrix} = \frac{-2c}{b \pm \sqrt{b^2 - 4ac}} \tag{3.13}$$

An illustration of the problem and the use of this alternative formula are provided in the following example.

EXAMPLE 3.7 Subtractive Cancellation

Problem Statement. Compute the values of the roots of a quadratic equation with $a = 1$, $b = 3000.001$, and $c = 3$. Check the computed values versus the true roots of $x_1 = -0.001$ and $x_2 = -3000$.

Solution. Figure 3.11 shows a Fortran 90 program that computes the roots x_1 and x_2 on the basis of the quadratic formula [(Eq. (3.12)]. Note that both single- and double-precision versions are given. Whereas the results for x_2 are adequate, the percent relative errors for x_1 are poor for the single-precision version, $\varepsilon_t = 2.4\%$. This level could be inadequate for many applied engineering problems. This result is particularly surprising because we are employing an analytical formula to obtain our solution!

The loss of significance occurs in the line of both programs where two relatively large numbers are subtracted. Similar problems do not occur when the same numbers are added.

On the basis of the above, we can draw the general conclusion that the quadratic formula will be susceptible to subtractive cancellation whenever $b^2 \gg 4ac$. One way to circumvent this problem is to use double precision. Another is to recast the quadratic formula in the format of Eq. (3.13). As in the program output, both options give a much smaller error because the subtractive cancellation is minimized or avoided.

FIGURE 3.11
Fortran 90 program to determine the roots of a quadratic.

```
PROGRAM fig0311
IMPLICIT none
REAL::a,b,c,d,x1,x2,x1r
DOUBLE PRECISION::aa,bb,cc,dd,x11,x22
a = 1.
b = 3000.001
c = 3.
d = SQRT(b * b - 4. * a * c)
x1 = (-b + d) / (2. * a)
x2 = (-b - d) / (2. * a)
PRINT *, 'Single-precision results:'
PRINT '(1x,a10,f20.14)', 'x1 = ', x1
PRINT '(1x,a10,f10.4)', 'x2 = ', x2
PRINT *
aa = 1.
bb = 3000.001
cc = 3.
dd = SQRT(bb * bb - 4. * aa * cc)
x11 = (-bb + dd) / (2. * aa)
x22 = (-bb - dd) / (2. * aa)
PRINT *, 'Double-precision results:'
PRINT '(1x,a10,f20.14)', 'x1 = ', x11
PRINT '(1x,a10,f10.4)', 'x2 = ', x22
PRINT *
PRINT *, 'Modified formula for first root:'
x1r = -2. * c / (b + d)
PRINT '(1x,a10,f20.14)', 'x1 = ', x1r
END

OUTPUT:
Single-precision results:
     x1 =      -.00097656250000
     x2 = -3000.0000
Double-precision results:
     x1 =      -.00100000000771
     x2 = -3000.0000
Modified formula for first root:
     x1 =      -.00100000000000
```

Note that, as in the foregoing example, there are times where subtractive cancellation can be circumvented by using a transformation. However, the only general remedy is to employ extended precision.

Smearing. Smearing occurs whenever the individual terms in a summation are larger than the summation itself. As in the following example, one case where this occurs is in series of mixed signs.

EXAMPLE 3.8 Evaluation of e^x using Infinite Series

Problem Statement. The exponential function $y = e^x$ is given by the infinite series

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

Evaluate this function for $x = 10$ and $x = -10$, and be attentive to the problems of round-off error.

Solution. Figure 3.12*a* gives a Fortran 90 program that uses the infinite series to evaluate e^x. The variable i is the number of terms in the series, *term* is the value of the current term added to the series, and *sum* is the accumulative value of the series. The variable *test* is the preceding accumulative value of the series prior to adding *term*. The series is terminated when the computer cannot detect the difference between *test* and *sum*.

Figure 3.12*b* shows the results of running the program for $x = 10$. Note that this case is completely satisfactory. The final result is achieved in 31 terms with the series identical to the library function value within seven significant figures.

Figure 3.12*c* shows similar results for $x = -10$. However, for this case, the results of the series calculation are not even the same sign as the true result. As a matter of fact, the negative results are open to serious question because e^x can never be less than zero. The problem here is caused by round-off error. Note that many of the terms that make up the sum are much larger than the final result of the sum. Furthermore, unlike the previous case, the individual terms vary in sign. Thus, in effect we are adding and subtracting large numbers (each with some small error) and placing great significance on the differences—that is, subtractive cancellation. Thus, we can see that the culprit behind this example of smearing is, in fact, subtractive cancellation. For such cases it is appropriate to seek some other computational strategy. For example, one might try to compute $y = e^{10}$ as $y = (e^{-1})^{10}$. Other than such a reformulation, the only general recourse is extended precision.

Inner Products. As should be clear from the last sections, some infinite series are particularly prone to round-off error. Fortunately, the calculation of series is not one of the more common operations in numerical methods. A far more ubiquitous manipulation is the calculation of inner products, as in

$$\sum_{i=1}^{n} x_i y_i = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n$$

This operation is very common, particularly in the solution of simultaneous linear algebraic equations. Such summations are prone to round-off error. Consequently, it is often desirable to compute such summations in extended precision.

(a) Program

```
PROGRAM fig0312
IMPLICIT none
REAL::term, test, sum,x
INTEGER::i
i = 0
term = 1.
sum = 1.
test = 0.
PRINT *, 'x = '
READ *, x
PRINT *, 'i', 'term', 'sum'
DO
  IF (sum.EQ.test) EXIT
  PRINT *, i, term, sum
  i = i + 1
  term = term*x/i
  test = sum
  sum = sum+term
END DO
PRINT *, 'exact value =',exp(x)
END
```

(b) Evaluation of e^{10}

```
x=
10
i        term              sum
0         1.000000          1.000000
1        10.000000         11.000000
2        50.000000         61.000000
3       166.666700        227.666700
4       416.666700        644.333400
5       833.333400       1477.667000
                  .
                  .
                  .
27   9.183693E-02      22026.420000
28   3.279890E-02      22026.450000
29   1.130997E-02      22026.460000
30   3.769989E-03      22026.470000
31   1.216126E-03      22026.470000
exact value =    22026.460000
```

(c) Evaluation of e^{-10}

```
x=
-10
i        term              sum
0         1.000000          1.000000
1       -10.000000         -9.000000
2        50.000000         41.000000
3      -166.666700       -125.666700
4       416.666700        291.000000
5      -833.333400       -542.333400
                  .
                  .
                  .
41  -2.989312E-09      8.137590E-05
42   7.117410E-10      8.137661E-05
43  -1.655212E-10      8.137644E-05
44   3.761845E-11      8.137648E-05
45  -8.359655E-12      8.137647E-05
exact value =    4.539993E-05
```

FIGURE 3.12
(a) A Fortran 90 program to evaluate e^x using an infinite series. (b) Evaluation of e^x. (c) Evaluation of e^{-x}.

Although the foregoing sections should provide rules of thumb to mitigate round-off error, they do not provide a direct means beyond trial and error to actually determine the effect of such errors on a computation. In the next chapter, we will introduce the Taylor series, which will provide a mathematical approach for estimating these effects.

PROBLEMS

3.1 Convert the following base-2 numbers to base-10: **(a)** 101101, **(b)** 101.101, and **(c)** 0.01101.

3.2 Compose your own program based on Fig. 3.9 and use it to determine your computer's machine epsilon.

3.3 In a fashion similar to that in Fig. 3.9, write a short program to determine the smallest number, $x_{\min}$, used on the computer you will be employing along with this book. Note that your computer will be unable to reliably distinguish between zero and a quantity that is smaller than this number.

3.4 The infinite series

$$f(n) = \sum_{i=1}^{n} \frac{1}{i^4}$$

converges on a value of $f(n) = \pi^4/90$ as n approaches infinity. Write a program in single precision to calculate $f(n)$ for $n = 10{,}000$ by computing the sum from $i = 1$ to 10,000. Then repeat the calculation but in reverse order—that is, from $i = 10{,}000$ to 1 using increments of -1. In each case, compute the true percent relative error. Explain the results.

3.5 Evaluate e^{-5} using two approaches

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

and

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

and compare with the true value of 6.737947×10^{-3}. Use 20 terms to evaluate each series and compute true and approximate relative errors as terms are added.

3.6 The derivative of $f(x) = 1/(1 - 3x^2)^2$ is given by

$$\frac{6x}{(1 - 3x^2)^2}$$

Do you expect to have difficulties evaluating this function at $x = 0.577$? Try it using 3- and 4-digit arithmetic with chopping.

3.7 **(a)** Evaluate the polynomial

$$y = x^3 - 7x^2 + 8x - 0.35$$

at $x = 1.37$. Use 3-digit arithmetic with chopping. Evaluate the percent relative error.

(b) Repeat **(a)** but express y as

$$y = ((x - 7)x + 8)x - 0.35$$

Evaluate the error and compare with part **(a)**.

3.8 Calculate the random access memory (RAM) in megabytes necessary to store a multidimensional array that is $20 \times 40 \times 120$. This array is double precision, and each value requires a 64-bit word. Recall that a 64-bit word = 8 bytes and 1 kilobyte = 2^{10} bytes. Assume that the index starts at 1.

3.9 Determine the number of terms necessary to approximate $\cos x$ to 8 significant figures using the Maclaurin series approximation

$$\cos x = 1 - \frac{x^2}{2} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \cdots$$

Calculate the approximation using a value of $x = 0.3\pi$. Write a program to determine your result.

3.10 Use 5-digit arithmetic with chopping to determine the roots of the following equation with Eqs. (3.12) and (3.13)

$$x^2 - 5000.002x + 10$$

Compute percent relative errors for your results.

3.11 How can the machine epsilon be employed to formulate a stopping criterion ε_s for your programs? Provide an example.