Numerical Analysis
by Patel ©1994
Saunders College Pub.

Chapter

1 NUMERICAL COMPUTATIONS

1.1 INTRODUCTION

In this text we are concerned with numerical methods used to solve the most common mathematical problems that arise in the physical, biological, and social sciences, and many other disciplines. The problem is stated in mathematical terms by using various assumptions. The next step is to solve the stated mathematical problem. Unfortunately, many practical problems do not have an analytical solution; consequently, we look for an approximation or numerical solution. Also, an analytical solution may not be convenient for numerical evaluation. Therefore, we look for methods that give an approximate solution to our formulated problem. Because these methods work with numbers and produce numbers, they are called numerical methods. Numerical analysis provides a means of proposing and analyzing numerical methods for the study and solution of mathematically stated problems. To facilitate computation, numerical methods are programmed for execution on a computer. A poorly written computer program can spoil a good numerical method both with inaccurate answers and by using excessive computer time for computation. Therefore, it is very important to take into account the programming aspects of a numerical method. A computer output must also be analyzed for its correctness.

A complete and unambiguous set of directions to solve a mathematical problem to the desired accuracy in a finite number of steps is called an **algorithm**. Thus a numerical method can be considered an algorithm.

We imagine a program library containing subroutines, written by experts, for every conceivable situation. In fact, there exists a large number of computer packages like IMSL (international mathematical and statistical library), NAG (numerical algorithm group), and more through which many subroutines are available on mainframe computers. Also, IMSL and NAG have special subsets of their full libraries for microcomputers. For microcomputers, MATHCAD, MATLAB, and MATHEMATICA also provide programs. Other packages are being developed continually. While it is easy to call these subroutines, there are many pitfalls in numerical computation. One should be able to recognize symptoms of numerical ill health and diagnose a problem. It is important to have a clear understanding of the numerical methods used by these subroutines.

In the next section, we develop some fundamental notions about digital computers since they are the principal means for our calculations.

1.2 NUMBER REPRESENTATION

As is well known, our usual number system is the decimal system. The number 796.85 is expressed as

$$796.85 = 7 \times 10^2 + 9 \times 10^1 + 6 \times 10^0 + 8 \times 10^{-1} + 5 \times 10^{-2}$$

The number 10 is the base of the decimal system.

Electrical impulses are either on or off and computers read pulses sent by their electrical components. If "off" state represents 0 and "on" state represents 1, then computers can use a system that needs only 0 and 1 as digits to represent a real number. This system is called the **binary system** and has base 2. Consider

$$\begin{aligned}(1001.101)_2 &= 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &\quad + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} \\ &= 8 + 0 + 0 + 1 + \frac{1}{2} + 0 + \frac{1}{8} \\ &= 9 + \frac{1}{2} + \frac{1}{8} = \frac{77}{8}\end{aligned}$$

Further, consider

$$\begin{aligned}(1101011111)_2 &= 1 \times 2^9 + 1 \times 2^8 + 0 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 \\ &\quad + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\ &= 512 + 256 + 0 + 64 + 0 + 16 + 8 + 4 + 2 + 1 = 863\end{aligned}$$

In order to represent 863, we need 10 binary digits. This is a major drawback of the binary system. The octal or hexadecimal number system (with base system 8) presents a compromise between the binary and decimal system when we discuss how numbers are stored in the computer. IBM 3033 uses the base 16 and the numbers 10, 11, 12, 13, 14, and 15 are usually denoted by A, B, C, D, E, and F, respectively. Most computers have an integer mode to represent integers and a floating-point mode to represent real numbers within given limits. The floating-point representation is closely connected to scientific notation. Letting x be any real number, x can be represented in floating-point form as

$$x = (\text{sign } x)(.d_1 d_2 \ldots d_t d_{t+1} \ldots)_\beta \times \beta^e \qquad \textbf{(1.2.1)}$$

where the characters d_i are digits in the base β system. In other words, $0 \leq d_i \leq \beta - 1$ for $i = 1, 2, \ldots$ with $d_1 \neq 0$ and e an integer. The number $(.d_1 d_2 \ldots d_t d_{t+1} \ldots)$ is called the **mantissa** and e is called the **exponent** or **characteristic** of x. Usually e is restricted by

$$-N \leq e \leq M \qquad \textbf{(1.2.2)}$$

for some large positive integers N and M. If during the calculations some computed quantity has an exponent $n > M$ then usually the result is meaningless and is called **overflow**. If an exponent $n < -N$, then usually the result is returned as zero without any warning message by many computers and is called **underflow**. Similarly an infinite representation of a mantissa cannot be used and so the mantissa of x has to be terminated at t digits. Let us denote this terminated number by $\text{fl}(x)$. This termination is done in

two ways. The first way is to delete the digits $d_{t+1}, d_{t+2}, \ldots$ to get the following **chopped** representation:

$$\text{fl}(x) = (\text{sign } x)(.d_1 d_2 \ldots d_t)_\beta \times \beta^e \tag{1.2.3}$$

The second way is to add $\beta/2$ to d_{t+1} and then chop off the resulting digits $\delta_{t+1}, \delta_{t+2}, \ldots$ to get the **rounded** representation

$$\text{fl}(x) = (\text{sign } x)(.\delta_1 \delta_2 \ldots \delta_t)_\beta \times \beta^{e_1} \tag{1.2.4}$$

where δ_i may or may not be d_i and e_1 may or may not be e.

EXAMPLE 1.1.1 The number 13/6 has an infinite decimal representation given by $13/6 = 2.166666\ldots = 0.2166666\ldots \times 10^1$. Letting $t = 5$, the chopping representation of 13/6 is

$$\text{fl}\left(\frac{13}{6}\right) = 0.21666 \times 10^1 = 2.1666$$

For the rounding representation, add 5 to the sixth digit, $6 + 5$, and then chop off the digits after the fifth digit. Thus

$$\text{fl}\left(\frac{13}{6}\right) = 0.21667 \times 10^1 = 2.1667 \qquad \blacksquare\blacksquare\blacksquare$$

The error that results from replacing a number by either its chopped representation or rounded representation is called **round-off error**.

In Table 1.2.1 the floating-point characteristics are given for commonly used digital computers (Atkinson 1989) for single precision.

Table 1.2.1

Computer	β	t	N	M	β^{1-t}
CDC 6600 & Cyber Series 170	2	48	975	1070	7.11×10^{-15}
Cray-1	2	48	16384	8191	7.11×10^{-15}
Hewlett Packard HP-45, 11C, 15C	10	10	98	100	1.00×10^{-9}
IBM 3033	16	6	64	63	9.54×10^{-7}
DEC VAX 11/780	2	24	128	127	1.19×10^{-7}
PDP-11	2	24	128	127	1.19×10^{-7}
Prime 850	2	23	128	127	2.38×10^{-7}

The question of rounding or chopping in Table 1.2.1 often depends on the installation or the compiler.

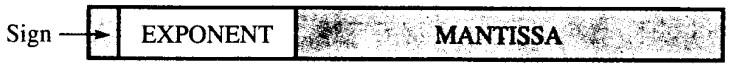
Let us denote the set of all numbers represented by Equation (1.2.3) or Equation (1.2.4) and zero by

$$F = F(\beta, t, N, M)$$

The real numbers, program instructions, integers, alphabetic symbols, and so on, are stored in words in digital computers. These words have a fixed number of digits.

The number of digits (bits) in a word is called the **word length**. For scientific calculations a long length is desirable; a short length, on the other hand, is significantly less expensive and perhaps more useful for business calculations and data processing. Word lengths range from 12 bits to 60 bits. In some computers a longer word is broken into smaller pieces called **bytes** (each consisting of 8 bits) for ease in handling.

Consider a hypothetical computer that uses 32 bits in a word. Of the 32 bits, the first bit is used to hold the sign of the number, 0 for + and 1 for −. The remaining 31 bits hold a binary number 0 to 1111111111 1111111111 1111111111 1. This applies only to integers. For a floating-point number, the first bit holds the sign, the next 7 bits hold the exponent (including one for the sign of the exponent), and the last 24 bits hold the mantissa.



Consider for example

1 1111001 1111111111 1111111111 1111

The first bit indicates that the number is negative. The next bit indicates that the exponent is negative. The next 6 bits, 111001, are equivalent to

$$1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 32 + 16 + 8 + 1 = 57$$

The last 24 bits indicate that the mantissa is

$$\begin{aligned} 1 \times 2^{-1} + 1 \times 2^{-2} + \cdots + 1 \times 2^{-24} &= \frac{1}{2}\left(1 + \frac{1}{2} + \cdots + \frac{1}{2^{23}}\right) \\ &= \frac{1}{2}\frac{\left(1 - \left(\frac{1}{2}\right)^{24}\right)}{\left(1 - \frac{1}{2}\right)} = 1 - \left(\frac{1}{2}\right)^{24} \\ &\approx 1 - (0.596046) \times 10^{-7} \\ &\approx 0.999999 \text{ to seven places} \end{aligned}$$

The closest 7 digit decimal number is $-0.6938893 \times 10^{-17}$ for $-(1 - (\frac{1}{2})^{24}) \times 2^{-57}$. Thus the machine number is used to represent any real number in the interval $(-0.69388935 \times 10^{-17}, -0.69388925 \times 10^{-17})$.

Since we are representing real numbers with approximate real numbers, our interest is to find the maximum error involved. Let x^* be an approximation of x in the decimal system for the following cases:

(1)	(2)	(3)
$x = 2.1666$	$x = 0.0004$	$x = 10000.0001$
$x^* = 2.1667$	$x^* = 0.0003$	$x^* = 10000.0000$
$\lvert x - x^* \rvert = 0.0001$	$\lvert x - x^* \rvert = 0.0001$	$\lvert x - x^* \rvert = 0.0001$

In all cases, the absolute error $|x - x^*|$ is 10^{-4}. In case (1) x^* is a good approximation for x, while in case (2) x is so small that $|x - x^*|$ represents a significant change. In case (3) x^* seems to be an excellent approximation. Thus it is clear that the ratio of $|x - x^*|$ to $|x|$ is important.

Let x^* be an approximation to x. The **relative error** in x^* is given by

$$\left|\frac{x - x^*}{x}\right| \quad \text{provided } x \neq 0$$

EXAMPLE 1.2.2

In case (1) the relative error $|(x - x^*)/x| = |(2.1666 - 2.1667)/2.1666| = 0.00005$; in case (2) the relative error $|(x - x^*)/x| = |(0.0004 - 0.0003)/0.0004| = 0.25$; and in case (3) the relative error $|(x - x^*)/x| = |(10000.0001 - 10000)/10000.0001| = 10^{-8}$.

In case (2) the relative error indicates that x^* is a poor approximation to x.

■ ■ ■

Since we do not know the true real number in a practical situation, we do not know what the error is. We will be happy with some bounds on the error. Let us find the relative error when we chop or round a given real number x in our decimal system. Let x be represented by

$$x = (\text{sign } x)(.d_1 d_2 \ldots d_t d_{t+1} \ldots) \times 10^e \tag{1.2.5}$$

We approximate x by simply chopping off the digits $d_{t+1}, d_{t+2}, \ldots$, to get the chopped representation

$$\text{fl}(x) = x^* = (\text{sign } x)(.d_1 d_2 \ldots d_t) \times 10^e \tag{1.2.6}$$

Thus the relative chopping error in x^* is given by

$$\left|\frac{x - x^*}{x}\right| = \frac{(.00 \ldots d_{t+1} d_{t+2} \ldots)}{(.d_1 d_2 \ldots d_t d_{t+1} \ldots)} = \frac{(.d_{t+1} d_{t+2} \ldots) \times 10^{-t}}{(.d_1 d_2 \ldots d_t d_{t+1} \ldots)}$$

Since $1 \leq d_1 \leq 9$, the minimum value of the denominator is 0.1 and $.d_{t+1} d_{t+2} \ldots < 1$, the relative chopping error is

$$\left|\frac{x - x^*}{x}\right| \leq \frac{10^{-t}}{0.1} = 10^{1-t} \tag{1.2.7}$$

For rounding, if $d_{t+1} < 5$, then $.d_{t+1} d_{t+2} \ldots \leq \frac{1}{2}$. Therefore,

$$\left|\frac{x - x^*}{x}\right| = \frac{(.d_{t+1} d_{t+2} \ldots) \times 10^{-t}}{(.d_1 d_2 \ldots d_t d_{t+1} \ldots)} \leq \frac{1}{2} 10^{1-t}$$

If $5 \leq d_{t+1} < 10$, then

$$\text{fl}(x) = x^* = (\text{sign } x) \times (.\delta_1 \delta_2 \ldots \delta_t) \times 10^{e_1}$$

where $e_1 = e$ or $e + 1$. If $e_1 = e$, then the relative rounding error in x^* is given by

$$\left|\frac{x - x^*}{x}\right| = \frac{|.d_1 d_2 \ldots d_t d_{t+1} \ldots - .\delta_1 \delta_2 \ldots \delta_t|}{(.d_1 d_2 \ldots d_t d_{t+1} \ldots)} \leq \frac{0.00 \ldots 05}{(0.1)} = \frac{1}{2} 10^{1-t}$$

If $e_1 = e + 1$, it can be proved that the relative rounding error in x^* is also given by

$$\left|\frac{x - x^*}{x}\right| \leq \frac{1}{2} 10^{1-t} \tag{1.2.8}$$

It can be proved for the base system β that the relative chopping error (Exercise 9) is

$$\left|\frac{x - x^*}{x}\right| \leq \beta^{1-t} \tag{1.2.9}$$

and the relative rounding error is

$$\left|\frac{x - x^*}{x}\right| \leq \frac{1}{2} \beta^{1-t} \tag{1.2.10}$$

In Table 1.2.1 the maximum relative chopping error in $\text{fl}(x)$ is given by the quantity β^{1-t}. Equations (1.2.9) and (1.2.10) depend only on the floating-point number system and the value of t. Therefore, they are independent of the size of the number. We may increase t to reduce round-off error. For instance, use of double precision instead of single precision reduces round-off error. The value of t is said to be the number of **significant digits** of a computer.

The smallest positive floating-point number ϵ, when added to the floating-point number 1.0 to produce a floating-point number different than 1.00, is known as the **machine epsilon**.

EXAMPLE 1.2.3 Let $x = 13/6$. For $t = 5$ and $\beta = 10$, the chopping representation of 13/6 is $\text{fl}(13/6) = 0.21666 \times 10^1$. From Equation (1.2.7)

$$\left|\frac{13/6 - \text{fl}(13/6)}{13/6}\right| \leq 10^{-4}$$

while $\left|\frac{13/6 - \text{fl}(13/6)}{13/6}\right| \approx 0.3 \times 10^{-4}$.

For $t = 5$, the rounding representation of 13/6 is $\text{fl}(13/6) = 0.21667 \times 10^1$. From Equation (1.2.8)

$$\left|\frac{13/6 - \text{fl}(13/6)}{13/6}\right| \leq \frac{1}{2} 10^{-4}$$

while $\left|\frac{13/6 - \text{fl}(13/6)}{13/6}\right| \approx 0.15 \times 10^{-4}$. ■■■

.3 FLOATING-POINT ARITHMETIC

In order to understand the kind of arithmetic done by computers, let us analyze computer arithmetic on a five-digit hypothetical machine that is similar to the actual arithmetic carried out by several common computers.

Let $x = 0.24689 \times 10^3$ and $y = 0.13579 \times 10^2$. These numbers are loaded in arithmetic registers. The arithmetic registers are capable of shifting numbers right and left in order to align the decimal point or to adjust the exponents during arithmetic operations. The length of an arithmetic register relative to a word length is an important characteristic of a computer. If the single precision number has t digits for its mantissa, then the arithmetic register should contain $t + p$ digits where p is comparable to t. This decision involves not just mathematical but economical factors. A register is called a single length register if $p = 0$, and a double length register if $p = t$. Several computer manufacturers use $p = 1$ because of economic considerations. This single extra digit is called a guard digit. The guard digit makes a noticeable effect on the accuracy of computed results. We will assume that our hypothetical machine has a double length register and uses chopping. In order to determine what values will be produced by a particular machine, one has to study the manuals supplied by a manufacturer.

Let us add x and y on our machine. The exponent of the smaller number is to be adjusted so that the exponents of both numbers are the same. Thus

$$x = 0.24689\,00000 \times 10^3$$
$$y = 0.01357\,90000 \times 10^3$$

These numbers are added in the accumulator and we get

$$x + y = 0.26046\,90000 \times 10^3$$

Since this number is to be stored, it is converted to the following five-digit floating-point number. Hence

$$\text{fl}(x + y) = 0.26046 \times 10^3$$

The result 0.26046×10^3 is stored as a computer word. Since the exact sum $= 0.260469 \times 10^3$, the relative error $|(x + y - \text{fl}(x + y))/(x + y)| = 0.345530 \times 10^{-4}$. Let us subtract y from x. In the accumulator

$$x - y = 0.23331\,10000 \times 10^3$$
$$\text{fl}(x - y) = 0.23331 \times 10^3$$

The result 0.23331×10^3 is stored.

Multiplication is simple because the exponents do not have to be aligned. Since

$$xy = 10^3 \times 10^2 \times 0.24689 \times 0.13579$$
$$= 0.335251931 \times 10^4$$

and $\text{fl}(xy) = 0.33525 \times 10^4$, the relative error $|(xy - \text{fl}(xy))/(xy)| = 0.57598 \times 10^{-5}$.

Division is not allowed when $y = 0$. If the mantissa of the numerator is greater than the mantissa of the denominator, we shift the mantissa of the numerator one place to the right. Thus

$$\frac{x}{y} = 10 \times \frac{0.24689}{0.13579}$$
$$= 10^2 \times \frac{0.0246890000}{0.1357900000} = 0.1818175000 \times 10^2$$

therefore $\text{fl}(x/y) = 0.18181 \times 10^2$ and the relative error $|((x/y) - \text{fl}(x/y))/(x/y)| = 0.41250 \times 10^{-4}$.

Normally $\text{fl}(x) \neq x$. Using Equations (1.2.9) and (1.2.10),

$$\frac{x - \text{fl}(x)}{x} = -\epsilon \tag{1.3.1}$$

where $-\beta^{1-t} \leq \epsilon \leq 0$ if chopped and $-\frac{1}{2}\beta^{1-t} \leq \epsilon \leq \frac{1}{2}\beta^{1-t}$ if rounded. Equation (1.3.1) can be expressed in a more useful form:

$$\text{fl}(x) = x(1 + \epsilon) \tag{1.3.2}$$

Denote machine addition, subtraction, multiplication, and division by the symbols $\oplus$, $\ominus$, $\otimes$, and $\oslash$ respectively. For any floating-point numbers x and y, we have from Equation (1.3.2)

$$\begin{aligned} \text{fl}(x + y) &= x \oplus y = (x + y)(1 + \epsilon_1) \\ \text{fl}(x - y) &= x \ominus y = (x - y)(1 + \epsilon_2) \\ \text{fl}(xy) &= x \otimes y = xy(1 + \epsilon_3) \end{aligned}$$

and

$$\text{fl}\left(\frac{x}{y}\right) = x \oslash y = \frac{x}{y}(1 + \epsilon_4) \tag{1.3.3}$$

where ϵ_1, ϵ_2, ϵ_3, and ϵ_4 may be different. It can be seen from Equations (1.3.2) and (1.3.3) that

$$\begin{aligned} \text{fl}(x + (y + z)) &= x \oplus (y \oplus z) = x \oplus (y + z)(1 + \epsilon_5) \\ &= (x + [(y + z)(1 + \epsilon_5)])(1 + \epsilon_6) \\ &= x(1 + \epsilon_6) + (y + z)(1 + \epsilon_5)(1 + \epsilon_6) \\ \text{fl}((x + y) + z) &= (x \oplus y) \oplus z = (x + y)(1 + \epsilon_7) \oplus z \\ &= [(x + y)(1 + \epsilon_7) + z](1 + \epsilon_8) \\ &= (x + y)(1 + \epsilon_7)(1 + \epsilon_8) + z(1 + \epsilon_8) \end{aligned}$$

Hence, often

$$x \oplus (y \oplus z) \neq (x \oplus y) \oplus z$$

In other words, the associative law breaks down. Similarly (Exercise 11) the distributive law often fails:

$$x \otimes (y \oplus z) \neq (x \otimes y) \oplus (x \otimes z)$$

EXAMPLE 1.3.1 Illustrate that the associative law breaks down. Let $x = 0.52867 \times 10^4$, $y = 0.38234 \times 10^2$, and $z = 0.25678 \times 10^1$. Find $x \oplus (y \oplus z)$ and $(x \oplus y) \oplus z$.

Since y and z are first added, the exponent of z is adjusted so that the exponents of y and z are the same. Hence

$$\begin{aligned} y &= 0.38234\,00000 \times 10^2 \\ z &= 0.02567\,80000 \times 10^2 \\ y + z &= 0.40801\,80000 \times 10^2 \end{aligned}$$

and

$$\text{fl}(y + z) = y \oplus z = 0.40801 \times 10^2$$

Now to add $(y \oplus z)$ to x, the exponents of $(y \oplus z)$ and x are adjusted. Therefore,

$$\begin{aligned} x &= 0.52867\,00000 \times 10^4 \\ y \oplus z &= 0.00408\,01000 \times 10^4 \\ x + (y \oplus z) &= 0.53275\,01000 \times 10^4 \end{aligned}$$

and

$$x \oplus (y \oplus z) = 0.53275 \times 10^4 \tag{1.3.4}$$

One can verify that

$$x \oplus y = 0.53249\,00000 \times 10^4$$

and

$$(x \oplus y) \oplus z = 0.53274 \times 10^4 \tag{1.3.5}$$

Comparing Equations (1.3.4) and (1.3.5), $x \oplus (y \oplus z) \neq (x \oplus y) \oplus z$. ■ ■ ■

One must not implicitly assume the validity of the associative law. Although the associative law for addition is not valid in floating-point arithmetic, it is comforting to know that the commutative law $x \oplus y = y \oplus x$ still holds and should be valuable in our programming.

Another important source of error is the subtraction of a number from a nearly equal number. Consider $x = \sqrt{457} \approx 0.2137755 \times 10^2$ and $y = \sqrt{456} \approx 0.2135415 \times 10^2$. Subtract y from x on the five-digit machine. First x and y would be stored as $\text{fl}(x) = 0.21377 \times 10^2$ and $\text{fl}(y) = 0.21354 \times 10^2$. Since our hypothetical machine has double length register,

$$\begin{aligned} \text{fl}(x) &= 0.21377\,00000 \times 10^2 \\ \text{fl}(y) &= 0.21354\,00000 \times 10^2 \\ \text{fl}(x) - \text{fl}(y) &= 0.00023\,00000 \times 10^2 \end{aligned}$$

and

$$\text{fl}(\text{fl}(x) - \text{fl}(y)) = \text{fl}(x) \ominus \text{fl}(y) = 0.23000 \times 10^{-1} = 0.02300$$

The last three zeros at the end of the mantissa are of no use. Since the exact value of $x - y \approx 0.000234 \times 10^2 = 0.0234$, we have the relative error

$$\left| \frac{x - y - \text{fl}(\text{fl}(x) - \text{fl}(y))}{x - y} \right| = 0.17170 \times 10^{-1} \tag{1.3.6}$$

This relative error is quite large when compared to the relative errors of $\text{fl}(x)$ and $\text{fl}(y)$. How can a more accurate result be obtained? Sometimes the problem can be reformulated to avoid the subtraction. In this example,

$$\begin{aligned} \sqrt{457} - \sqrt{456} &= \frac{(\sqrt{457} - \sqrt{456})(\sqrt{456} + \sqrt{457})}{\sqrt{457} + \sqrt{456}} \\ &= \frac{1}{\sqrt{457} + \sqrt{456}} \approx 1 \oslash (\text{fl}(x) + \text{fl}(y)) \\ &= 1 \oslash (0.42731 \times 10^2) = 0.23402 \times 10^{-1} = 0.023402 \end{aligned}$$

The relative error 0.72643×10^{-5} is very small compared to Equation (1.3.6).

EXERCISES

1. Express the following base β numbers in floating-point form.
   (a) $(123.456)_{10}$ (b) $(27.653)_8$ (c) $(10101.1101)_2$ (d) $(AB.168)_{16}$
2. Express the following base β numbers in decimal form.
   (a) $(10.001101)_2$ (b) $(0.775)_8$ (c) $(1.89ABC)_{16}$ (d) $(67.015)_8$
3. Write the elements of the set $F(2, 3, 1, 2)$. Convert the elements in the decimal system and then represent the numbers as points on a straight line.
4. Convert $(0.1)_{10}$ to binary form. Are ten steps of length $(0.1)_{10}$ in binary form the same as one step of length 1.0? (*Hint:* $\frac{1}{10} = \frac{0}{2} + \frac{0}{2^2} + \frac{0}{2^3} + \frac{1}{2^4} \cdots$.)
5. Let $\text{fl}(x)$ be given by chopping. Prove that (a) $\text{fl}(-x) = -\text{fl}(x)$ and (b) $\text{fl}(\beta^r x) = \beta^r \text{fl}(x)$ by assuming that underflow or overflow does not occur.
6. If the following approximations are used, find the absolute error and relative error.
   (a) $\frac{1}{7} \approx 0.14$ (b) $\frac{1}{7} \approx 0.1428$
7. How many significant digits are there if (a) 852.045 is approximated by 852.01, (b) 0.000452 is approximated by 0.00041, and (c) 1.2345 is approximated by 1.234?
8. Find the absolute error and the relative error for Exercise 7.
9. Prove Equations (1.2.9) and (1.2.10).
10. Let $x = 0.5678 \times 10^1$, $y = 0.3456 \times 10^2$, and $z = 0.1234 \times 10^3$. Perform the following operations on a hypothetical machine that uses four-digit floating-point arithmetic, double precision accumulator, and chops the resulting number to four digits before any subsequent operation is performed.
    (a) $x \oplus y$ (b) $y \oplus x$ (c) $(x \oplus y) \oplus z$ (d) $x \oplus (y \oplus z)$
    (e) $x \otimes (y \oplus z)$ (f) $(x \otimes y) \oplus (x \otimes z)$
11. Prove that $x \otimes (y \oplus z) \neq (x \otimes y) \oplus (x \otimes z)$ in some cases.
12. Consider a machine that uses four-digit floating-point arithmetic. Let $x = \ln 2.10 = 0.74194$, and $y = \ln 2.11 = 0.74669$. Compute $\text{fl}(x) \ominus \text{fl}(y)$ and determine its relative error.
13. Indicate how the following formulas should be written to avoid the loss of significant digits due to subtraction:
    (a) $\ln x - \ln y$ (b) $e^x - x - 1$ if x is close to 0 (c) $1 - \cos x$ if x is close to 0 (d) e^{x-y}
    (e) $\dfrac{-b + \sqrt{b^2 - 4ac}}{2a}$ if $b > 0$ and $\dfrac{-b - \sqrt{b^2 - 4ac}}{2a}$ if $b < 0$
14. Consult your computer manuals and indicate whether your computer has a single length register, a double length register, a register with a guard digit, or none of the above. Write a brief description of the addition operation.
15. Find out the base and number of digits in the mantissa of your computer. What is the largest number and the smallest number represented by your computer?

COMPUTER EXERCISE

C.1. Write a program to find the machine epsilon of your computer.

1.4 NUMERICAL COMPUTING

In order to solve a problem on a digital computer, we perform the following steps:

Step 1 Construct a mathematical model for a given problem by using a variety of simplifying assumptions. As a consequence of these assumptions, the resulting mathematical model has inherent limitations. Be aware that a numerical solution of a mathematical model cannot improve the accuracy of a mathematical model except by coincidence. Since we cannot judge a priori the error due to either the mathematical model or the numerical method, we need to find as accurate a numerical solution of a mathematical model as possible. This mathematical model is a set of formulas, inequalities, equations and so on.

Step 2 Decide what mathematical or numerical method to use to solve a mathematical problem. Our concern is to find the best numerical method for solving a given mathematical problem. How should a numerical method be evaluated? Two important properties come to mind—accuracy and efficiency. The amount of computer time used in the execution of a particular method can be used to measure the efficiency. The time it takes to complete floating-point arithmetic is usually much longer than the time it takes to store, test, perform integer arithmetic, fetch, and so on. Also, there are comparatively few nonarithmetic steps in most numerical algorithms, so a reasonable estimate of the required time can be obtained by counting the number of required additions, subtractions, multiplications, and divisions. Because addition and subtraction take approximately the same amount of time on nearly all computers, these two operations will be counted together. Division time is longer than multiplication time for one operation. We can use an expression

$$E = pA + qM + rD \tag{1.4.1}$$

where p, q, and r are the total number of floating-point additions, multiplications, and divisions respectively in a given algorithm and A, M, and D are the amounts of time it takes a computer to perform one addition, one multiplication, and one division respectively. If many algorithms with equal accuracy are available, then the algorithm with the smallest value of E is preferred.

The computation times in seconds for 1000 repetitions of common arithmetic operations on an IBM PC are given in Table 1.4.1. The installation of a mathematical coprocessor on an IBM PC or a compatible machine increases the speed of floating-point arithmetic considerably. For comparison, the computation times with and without the mathematical coprocessor Intel 8087 using Turbo Pascal on an IBM PC (Lastman and Sinha 1988) are given in Table 1.4.1.

Table 1.4.1

Operation	Time without Intel 8087	Time with Intel 8087
Addition	0.28	0.17
Subtraction	0.28	0.17
Multiplication	1.04	0.17
Division	1.71	0.22

The closeness of a computed answer to the exact answer is the accuracy of the computed answer. By their very nature, in most cases numerical methods do not give exact answers. There are many sources that contribute to the inaccuracy of the answers produced by a numerical method.

1. **Input error:** Data from measurements of practical problems contain errors that affect the accuracy of calculations based on the data.
2. **Round-off error:** A computer has a fixed word length and therefore most numbers, including those obtained by arithmetic operations, cannot be expressed exactly. Each number is represented by its nearest machine number. This type of error is called round-off error and is a characteristic of the computer or the computer language. The subtraction of two nearly equal numbers can be avoided by reformulating the problem. Thus the resulting vast increase in round-off error can be avoided. The generated round-off error contaminates subsequent calculations and the error propagation becomes an important source of error.
3. **Propagated error:** In order to see how errors accumulate in a complicated algorithm, consider the sum of two positive numbers x_1 and x_2. The first step is to convert these numbers into floating-point numbers by chopping or rounding. Errors are thus introduced. Denoting the converted numbers with an overbar and using Equation (1.3.2), we get

$$\bar{x}_1 = \text{fl}(x_1) = x_1(1 + \epsilon_1) \quad \text{and} \quad \bar{x}_2 = \text{fl}(x_2) = x_2(1 + \epsilon_2) \tag{1.4.2}$$

where $|\epsilon_i| \leq \mu$ for $i = 1$ and 2.

When we add these two numbers, we actually compute $\bar{x}_1 \oplus \bar{x}_2$ (recall $\oplus$ denotes machine addition). The actual error is given by

$$\begin{aligned}(x_1 + x_2) - (\bar{x}_1 \oplus \bar{x}_2) &= [(x_1 + x_2) - (\bar{x}_1 + \bar{x}_2)] + [(\bar{x}_1 + \bar{x}_2) - (\bar{x}_1 \oplus \bar{x}_2)] \\ &= [(x_1 - \bar{x}_1) + (x_2 - \bar{x}_2)] + [(\bar{x}_1 + \bar{x}_2) - (\bar{x}_1 \oplus \bar{x}_2)]\end{aligned} \tag{1.4.3}$$

The term in the first brackets on the right-hand side of Equation (1.4.3) is the error propagated because of the initial conversion to floating-point numbers. Consequently, this error is called the propagated error. The term in the second brackets on the right-hand side of Equation (1.4.3) is the error generated because of the machine's arithmetic by rounding or chopping. It is called the round-off error. Using Equation (1.3.3), we have

$$\bar{x}_1 \oplus \bar{x}_2 = (\bar{x}_1 + \bar{x}_2)(1 + \epsilon_3) \tag{1.4.4}$$

where $|\epsilon_3| \leq \mu$. Using Equations (1.4.2) and (1.4.4) in Equation (1.4.3), we get

$$|(x_1 + x_2) - (\bar{x}_1 \oplus \bar{x}_2)| \leq |-\epsilon_1 x_1 - \epsilon_2 x_2| + |-\epsilon_3(\bar{x}_1 + \bar{x}_2)| \tag{1.4.5}$$

Further, using Equation (1.4.2), the last term of Equation (1.4.5) simplifies and is given by

$$\begin{aligned}\epsilon_3(\bar{x}_1 + \bar{x}_2) &= \epsilon_3(x_1 + \epsilon_1 x_1 + x_2 + \epsilon_2 x_2) \\ &= \epsilon_3(x_1 + x_2) + \epsilon_3(\epsilon_1 x_1 + \epsilon_2 x_2)\end{aligned}$$

Using this and $|\epsilon_i| \leq \mu$ for $i = 1$, 2, and 3 in Equation (1.4.5), we get

$$|(x_1 + x_2) - (\bar{x}_1 \oplus \bar{x}_2)| \leq (2\mu + \mu^2)(x_1 + x_2) \tag{1.4.6}$$

Since $|\epsilon_i| \leq \mu$, we replaced $|\epsilon_i|$ by μ in Equation (1.4.5). Thus Equation (1.4.6) gives the error bound for the worst case that may be encountered. When large numbers of operations are involved, the bounds may be several times larger than the actual error involved and therefore are too pessimistic. Often, the round-off errors tend to cancel each other, but, under the right circumstances, the round-off errors can grow like a rolling snowball. This phenomenon is referred to as instability and will be treated later.

EXAMPLE 1.4.1 Add $x_1 = 0.36789$ nad $x_2 = 2.5678$ using four-digit floating-point arithmetic with chopping.

Our machine transforms these numbers as $\bar{x}_1 = 0.3678 \times 10^0$ and $\bar{x}_2 = 0.2567 \times 10^1$. For adding, we have

$$\bar{x}_1 = 0.0367\,8000 \times 10^1 \quad \text{and} \quad \bar{x}_2 = 0.2567\,0000 \times 10^1$$

Then

$$\bar{x}_1 + \bar{x}_2 = 0.2934\,8000 \times 10^1 \quad \text{and so} \quad \bar{x}_1 \oplus \bar{x}_2 = 0.2934 \times 10^1$$

Since the exact sum $x_1 + x_2 = 2.93569$, the exact absolute error $|x_1 + x_2 - (\bar{x}_1 \oplus \bar{x}_2)| = 0.00169$. The propagated error is $|x_1 + x_2 - (\bar{x}_1 + \bar{x}_2)| = |(x_1 - \bar{x}_1)| + (x_2 - \bar{x}_2)| = |2.93569 - 2.93480| = 0.00089$, and the round-off error is $|\bar{x}_1 + \bar{x}_2 - (\bar{x}_1 \oplus \bar{x}_2)| = |2.9348 - 2.934| = 0.0008$.

Since we are chopping, $\mu = \beta^{1-t} = 10^{-3} = 0.001$. Hence, from Equation (1.4.6)

$$|(x_1 + x_2) - (\bar{x}_1 \oplus \bar{x}_2)| \leq (2\mu + \mu^2)(x_1 + x_2) = 0.00587$$

Thus

$$|(x_1 + x_2) - (\bar{x}_1 \oplus \bar{x}_2)| = 0.00169 < 0.00587$$

is true but not very accurate. ■ ■ ■

4. **Truncation error:** This error occurs when we truncate the process after a certain number of steps because, for an exact result, an infinite sequence of steps or too many steps was required. For example, Maclaurin's series for e^x is given by

$$e^x = 1 + x + \frac{x^2}{2!} + \cdots + \frac{x^n}{n!} + \cdots \tag{1.4.7}$$

Equation (1.4.7) can be written as the iteration

$$r_i = r_{i-1} + \frac{x^i}{i!} \quad \text{for } i = 1, 2, \ldots$$

with

$$r_0 = 1 \tag{1.4.8}$$

This formulation becomes more accurate as we use more and more iterations. However, because of computational resources, a decision must be made to stop the iterative process. Truncation error is the error introduced when we truncate

the process after a certain finite number of steps on a hypothetical perfect computer that had no round-off errors (all digits were retained). Since we do not have a perfect computer, more iterations mean more arithmetic operations, and more arithmetic operations mean more round-off errors and more propagated errors. In many cases, reducing truncation error means increasing round-off error.

Let us find $e^{1/2}$ using Equation (1.4.8). Since we cannot go on for ever, we stop somewhere. For simplicity, let us stop at $i = 3$. Then

$$r_3 = r_2 + \frac{1}{3!}\left(\frac{1}{2}\right)^3 = r_1 + \frac{1}{2!}\left(\frac{1}{2}\right)^2 + \frac{1}{3!}\left(\frac{1}{2}\right)^3$$

$$= r_0 + \frac{1}{1!}\left(\frac{1}{2}\right) + \frac{1}{2!}\left(\frac{1}{2}\right)^2 + \frac{1}{3!}\left(\frac{1}{2}\right)^3$$

We approximate $e^{1/2}$ by r_3. Thus

$$e^{1/2} \approx r_3 = 1 + \frac{1}{1!}\left(\frac{1}{2}\right) + \frac{1}{2!}\left(\frac{1}{2}\right)^2 + \frac{1}{3!}\left(\frac{1}{2}\right)^3 = \frac{79}{48}$$

An infinite sequence of steps is required for the exact result. By stopping at $i = 3$ we introduce a truncation error given by (Theorem A.8)

$$tr = \left(\frac{1}{2}\right)^4 \frac{1}{4!} e^{\xi}$$

where ξ is between 0 and 1/2.

Since ξ is not known, we estimate the maximum value of e^{ξ} on the closed interval [0, 1/2]. Thus

$$|tr| \leq \frac{e^{1/2}}{2^4\, 4!} \approx 0.00429$$

The actual truncation error $= e^{1/2} - (79/48) \approx 0.00289 < 0.00429$.

Step 3 After selecting a numerical method, we must carry out the programming of our numerical method. It is expected that the student is familiar with the rudiments of programming. The programming language could be Basic, Fortran, Pascal, C, or any other language.

A program written for a particular set of numbers must be rewritten for another set of numbers. In most cases, few additional statements are required to write a program to handle a general case. It will be worth the extra effort. Write out the mathematical algorithm in complete detail and write the code in a style that is easy to read and understand. Check your code thoroughly for omissions and errors before heading for a terminal. Do not rush. It pays to trace through the code with pencil and paper on a typical and simple example.

Always print the input data and initially print intermediate results to understand the program's operations. It helps to have labels for the output. Write a long program in steps by writing and testing a series of subroutines and function subprograms.

Use comments so that another person can understand what the code does. Insert blank comment lines in order to improve the readability of the code. For each subroutine, explain through comments the parameters used.

Step 4 The chance of a human being making an arithmetic error is very high, while the chance of a machine making an arithmetic error is very low. The main concern is programming error. Fortunately, some programming errors are repeated many times during the execution of a program and therefore the existence of those errors are identified by absurd numerical output. For a complex and lengthy computer program, it becomes difficult to detect and correct small but important logical programming errors. This makes debugging a very important component of the computer programming process and sometimes it makes a crucial difference in the numerical results. Therefore, it is extremely important to test a computer program for known results, since a poorly written computer program can spoil an excellent method both by providing inaccurate results and by using excessive computer time.

In this text we will discuss numerical methods and analyze them. It is assumed that the reader will write computer programs for these methods.

EXERCISES

1. Prove that $|x_1x_2 - \bar{x}_1 \otimes \bar{x}_2| \leq |x_1x_2|(3\mu + 3\mu^2 + \mu^3)$ where x_1 and x_2 are real numbers and

$$\mu = \begin{cases} \beta^{1-t} & \text{if chopped} \\ \frac{1}{2}\beta^{1-t} & \text{if rounded} \end{cases}$$

2. Let $x_1 = 0.12345 \times 10^2$ and $x_2 = 0.23456 \times 10^1$. Use four-digit chopped floating-point arithmetic to find $\bar{x}_1 \otimes \bar{x}_2$. Find the absolute propagated and round-off errors. Compare the exact error with the upper error bound given in Exercise 1.
3. Add $x_1 = 0.12345 \times 10^2$ and $x_2 = 0.23456 \times 10^1$ using four-digit chopped floating-point arithmetic. Find the absolute propagated and round-off errors. Compare the exact error with the upper error bound given by Equation (1.4.6).
4. Prove that

$$|(((x_1 + x_2) + x_3) + x_4) - (((\bar{x}_1 \oplus \bar{x}_2) \oplus \bar{x}_3) \oplus \bar{x}_4)| \leq \mu(4x_1 + 4x_2 + 3x_3 + 2x_4) + \mu^2(6x_1 + 6x_2 + 3x_3 + x_4) + \mu^3(4x_1 + 4x_2 + x_3) + \mu^4(x_1 + x_2)$$

where x_1, x_2, x_3, and x_4 are positive real numbers and

$$\mu = \begin{cases} \beta^{1-t} & \text{if chopped} \\ \frac{1}{2}\beta^{1-t} & \text{if rounded} \end{cases}$$

The first term on the right-hand side of the inequality contributes significantly. In that term x_1 is multiplied by 4 while x_4 is multiplied by 2. This and other terms on the right-hand side of the inequality suggest that the best strategy for addition is to add from the smallest to the largest.

5. Add $x_1 = 0.36789$, $x_2 = 2.5678$, $x_3 = 0.12345$, and $x_4 = 0.034567$ using four-digit chopped floating-point arithmetic. Rearrange these numbers from the smallest to the largest and then add them using four-digit chopped floating-point arithmetic. Compare these sums with the exact sum.

SUGGESTED READINGS

K. E. Atkinson, *An Introduction to Numerical Analysis,* 2nd ed., John Wiley & Sons, New York, 1989.

G. J. Lastman and N. K. Sinha, *Microcomputer-Based Numerical Methods for Science and Engineering,* Saunders College Publishing, Philadelphia, 1989.