1.2 ERRORS IN COMPUTATIONS

In analyzing the accuracy of numerical results, the numerical analyst should be aware of the possible sources of error in each stage of the computational process and of the extent to which these errors can affect the final answer. There are three types of errors that occur in a computation. First, there are errors that we call "initial data" errors. When the equations of the mathematical model are formed, these errors arise because of idealistic assumptions made to simplify the model, inaccurate measurements of data, miscopying of figures, or the inaccurate representation of mathematical constants (for example, if the constant π occurs in an equation, we must replace π by 3.1416 or 3.141593, etc.). Another class of errors, "truncation" errors, occurs when we are forced to use mathematical techniques that give approximate, rather than exact, answers. For example, suppose we use the Maclaurin's series expansion to represent e^x so that $e^x = 1 + x + x^2/2! + \cdots + x^n/n! + \cdots$. If we want a number that approximates e^β for some β, we must terminate the expansion in order to obtain $e^\beta \approx 1 + \beta + \beta^2/2! + \cdots + \beta^k/k!$. Thus $e^\beta = 1 + \beta + \beta^2/2! + \cdots + \beta^k/k! + E$ where E is the truncation error introduced in the calculation. Truncation errors in numerical analysis usually occur because many numerical methods are iterative in nature, with the approximations theoretically becoming more accurate as we take more iterations. As a practical matter, we must stop the iteration after a finite number of steps, and thus introduce a truncation error. The last type of error we shall consider, "round-off" or "rounding" errors, is due to the fact that a computer has a finite word length. Thus most numbers and the results of arithmetic operations on most numbers cannot be represented exactly on a computer. Even though the computer is capable of representing numerical values and performing operations on them, we should be aware of how this is accomplished so that we can understand the error that is produced by inexact representation.

Initial data errors and truncation errors are dependent mostly on the particular problem we are examining, and we shall deal with them as they arise in the context of the different numerical methods we derive throughout the text. The total effect of round-off errors is sometimes dependent on the particular problem in the sense that the more operations we perform, the more we can probably expect the round-off error to affect the solution. The individual round-off error caused by any individual number representation or arithmetic operation is dependent, however, on the particular computer being used; and thus we shall examine the possible sources of this error in the next section before we introduce any specific numerical methods. We emphasize that our ultimate concern is the effect of total error from any and all sources. For example, we shall later see problems in which a "small" error (no matter from what source) can cause a "large" error in the final solution. Problems of this type are called *ill-conditioned* and must be treated very carefully to obtain an acceptable computed answer.

There are two ways to measure the size of errors. In analyzing the error of a computation, if we let $\bar{x}$ represent the "computed approximation" to the "true solution" x, then we define the *absolute error* to be $(x - \bar{x})$ and the *relative error* to be $(x - \bar{x})/x$. (If the true solution x is zero, then we say the relative error is undefined.) We shall consider these concepts again in later sections, but we briefly pause to mention here that the relative error is usually more significant than the absolute error, and hence we shall try to establish bounds for the relative error whenever possible. To illustrate this point, suppose that in "Computation A" we have $x = 0.5 \times 10^{-4}$ and $\bar{x} = 0.4 \times 10^{-4}$; in "Computation B" we have $x = 5000$ and $\bar{x} = 4950$. The absolute errors are 0.1×10^{-4} and 50, respectively; but the relative errors are 0.2 and 0.01, respectively. Stated differently, Computation A has a 20% error; Computation B has only a 1% error.

In investigating the effect of the total error in various methods, we shall often mathematically derive an "error bound," which is a limit on how large the error can be. (This limit applies to both absolute and relative errors.) It is important that the reader realize that the error bound can be much larger than the actual error and in practice often is. Any mathematically derived error bound must account for the worst possible case that can occur and is often based upon certain simplifying assumptions about the problem, assumptions which in many particular cases cannot be actually tested. For the error bound to be used in any practical way, the user must have a good understanding of how the error bound was derived in order to know how crude it is; i.e., how likely it is to overestimate the actual error. Of course, whenever possible, our goal is to eliminate or lessen the effects of errors, rather than to estimate them after they occur.

1.3 ROUNDING ERRORS AND FLOATING-POINT ARITHMETIC

The first type of rounding error that we encounter in performing a computation evolves from the fact that most real numbers cannot be represented exactly on a computer. Readers are probably not surprised by this statement since they are aware that irrational numbers such as π or e have an infinite nonrepeating decimal expansion. Thus they know that even for a hand computation they must use an approximation such as 3.14159 for π or 2.718 for e, and they carry as many digits in their approximations as they feel are necessary for a particular computation. Nevertheless, they realize that once these approximations have been used, an error that can never exactly be corrected has been introduced into the calculation.

Since only a finite number of digits can be represented in computer memory, each number x must be represented in some fashion that uses only a fixed number of digits. One of the most common forms is the "floating-point" form

in which one position is used to identify the sign of x, a prescribed number of digits are used to represent the "mantissa" or fractional form of x, and an integer is used to represent the "exponent" or "characteristic" of x with respect to the base b of the representation. (Modern, preferred terminology uses "significand" for mantissa and "exrad" for exponent.) Thus each x can be thought of as being represented by a number $\bar{x}$ of the form $\pm(0.a_1a_2 \ldots a_m) \times (b^c)$ where m is the number of digits allowed in the mantissa, b is the base of the representation, and c is the exponent. Additionally there are two machine-dependent constants, μ and M, such that $\mu \leq c \leq M$. We shall consider three bases: (i) $b = 10$ (decimal), with which the reader is familiar and which is used on some machines; (ii) $b = 16$ (hexadecimal), which is common to the IBM 360 and 370 series; and (iii) $b = 2$ (binary), which is, in a sense, the most fundamental of the three. Proper form requires the mantissa, $a \equiv 0.a_1a_2 \ldots a_m$, to satisfy $|a| < 1 = b^0$ and each a_i, $1 \leq i \leq m$, to be an integer such that $0 \leq a_i \leq b - 1$ with $a_1 \neq 0$ (unless $\bar{x} = 0$). The floating-point representation, $\bar{x}$, is then said to be "normalized."

The floating-point decimal form ($b = 10$) should be familiar to the reader. For example, the decimal number 150.623 is the same as 0.150623×10^3 and can also be regarded as

$$(1 \times 10^2) + (5 \times 10^1) + (0 \times 10^0) + (6 \times 10^{-1}) + (2 \times 10^{-2}) + (3 \times 10^{-3}).$$

Likewise, the binary number 110.011 equals

$$(1 \times 2^2) + (1 \times 2^1) + (0 \times 2^0) + (0 \times 2^{-1}) + (1 \times 2^{-2}) + (1 \times 2^{-3}),$$

and the hexadecimal number 15F.A03 equals

$$(1 \times 16^2) + (5 \times 16^1) + (\mathrm{F} \times 16^0) + (\mathrm{A} \times 16^{-1}) + (0 \times 16^{-2}) + (3 \times 16^{-3}).$$

Note that there are only two digits, 0 and 1, in the binary system; and there are sixteen digits in the hexadecimal system; 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F where the decimal equivalents of A, B, C, D, E, F are 10, 11, 12, 13, 14, 15, respectively. Also note that the hexadecimal system is a natural extension of the binary system since $2^4 = 16$, and hence there is precisely one hexadecimal digit for each group of four binary digits ("bits") and vice versa [$0 = (0000)_2$, $7 = (0111)_2$, $\mathrm{A} = (1010)_2$, $\mathrm{F} = (1111)_2$, etc.].

The conversion of an integer from one system to another is fairly simple and can probably best be presented in terms of an example. Let $k = 275$ in decimal form; that is, $k = (2 \times 10^2) + (7 \times 10^1) + (5 \times 10^0)$. Now $(k/16^2) > 1$, but $(k/16^3) < 1$; so in hexadecimal form k can be written as $k = (\alpha_2 \times 16^2) + (\alpha_1 \times 16^1) + (\alpha_0 \times 16^0)$. Now $275 = 1(16^2) + 19 = 1(16^2) + 1(16) + 3$; and so the decimal integer 275 can be written in hexadecimal form as 113; that is, $(275)_{10} = (113)_{16}$. The reverse process is even simpler. For example, $(5\mathrm{C}3)_{16} = 5(16^2) + 12(16) + 3 = 1280 + 192 + 3 = (1475)_{10}$. Conversion of a hexadecimal fraction to a decimal is similar. For example, $(0.2\mathrm{A}8)_{16} = (2/16) + (\mathrm{A}/16^2) + (8/16^3) = (2(16^2) + 10(16) + 8)/16^3 = (680)/4096 = (0.166)_{10}$ (carrying only three digits in

the decimal form). Conversion of a decimal fraction to hexadecimal (or binary) proceeds as in the following example. Consider the number $r_1 = 1/10 = 0.1$ (decimal form). Then there exist constants $\{\alpha_k\}_{k=1}^{\infty}$ such that

$$r_1 = 0.1 = \alpha_1/16 + \alpha_2/16^2 + \alpha_3/16^3 + \alpha_4/16^4 + \cdots.$$

Now $16r_1 = 1.6 = \alpha_1 + \alpha_2/16 + \alpha_3/16^2 + \alpha_4/16^3 + \cdots$. Thus $\alpha_1 = 1$ and $r_2 \equiv 0.6 = \alpha_2/16 + \alpha_3/16^2 + \alpha_4/16^3 + \cdots$. Again $16r_2 = 9.6 = \alpha_2 + \alpha_3/16 + \alpha_4/16^2 + \cdots$, so $\alpha_2 = 9$ and $r_3 \equiv 0.6 = \alpha_3/16 + \alpha_4/16^2 + \cdots$. From this stage on we see that the process will repeat itself, and so we have $(0.1)_{10}$ equals the infinitely repeating hexadecimal fraction, $(0.1999 \ldots)_{16}$. Since $1 = (0001)_2$ and $9 = (1001)_2$, we also have the infinite binary expansion

$$r_1 = (0.1)_{10} = (0.1999 \ldots)_{16} = (0.0001\ 1001\ 1001\ 1001 \ldots)_2.$$

From the example above we begin to discern one problem of number representation. Not only do we have problems with irrational numbers and infinite repeating decimal expansions such as $1/3 = 0.333 \ldots$, but also we see that an m-digit terminating fraction with respect to one base may not have an n-digit terminating representation in another base. (If we were performing an iteration on a hexadecimal machine, the example above suggests that we should probably choose a step size of $h = 1/16$ over $h = 1/10$, $h = 1/1024 = 1/2^{10}$ over $h = 1/1000$, etc., if at all possible.) In Table 1.1 we have taken several integers k, formed the reciprocals, $1/k$, and added the reciprocal to itself k times. The theoretical result of each computation should, of course, equal 1. The calculations were performed on a six-digit hexadecimal machine.†

TABLE 1.1

k	Sum($1/k$)	k	Sum($1/k$)	k	Sum($1/k$)
2	0.1000000E 01	9	0.9999999E 00	16	0.1000000E 01
3	0.9999999E 00	10	0.9999996E 00	⋮	
4	0.1000000E 01	11	0.9999997E 00	1000	0.9999878E 00
5	0.9999999E 00	12	0.9999998E 00	1006	0.9999912E 00
6	0.9999998E 00	13	0.9999999E 00	1012	0.9999843E 00
7	0.9999999E 00	14	0.9999995E 00	1018	0.9999678E 00
8	0.1000000E 01	15	0.9999999E 00	1024	0.1000000E 01

We must, of course, realize that part of the error in Table 1.1 is due to the round-off from addition. We shall discuss this error momentarily, but for now the reader should notice the relative accuracies for the values of k that are powers of 2. For example, compare $k = 1000$ to $k = 1024 = 2^{10}$.

We next consider how numbers are represented in an m-digit machine, particularly those numbers for which m digits are not sufficient to represent the

†Most of the computer programs in this text were run on an IBM 370/158.

number exactly. For example, how might a number like $\frac{1}{3}$ be represented on a 5-digit decimal computer since $\frac{1}{3} = 0.3333333\ldots$. As a general example, suppose a real number x is given exactly by

$$x = \pm(0.\tilde{a}_1\tilde{a}_2 \ldots \tilde{a}_m\tilde{a}_{m+1}\ldots) \times 10^c, \qquad \tilde{a}_1 \neq 0$$

and suppose we want to represent x in an m-digit decimal computer. There are two common ways of representing x. First, we can simply let $a_k = \tilde{a}_k$, $1 \leq k \leq m$, discard the remaining digits, and let the computer representation be

$$\bar{x} = \pm(0.a_1a_2 \ldots a_m) \times 10^c.$$

This method is known as "chopping." The other representation is the familiar "symmetric rounding" process, which is equivalent to adding $5 \times 10^{c-m-1}$ to x and then chopping (we think of adding 5 to $\tilde{a}_{m+1}$). Of course, if $c < \mu$ or $c > M$, the number lies outside the range of admissible computer representations. If $c < \mu$, (underflow), it is common to regard x as zero, notify the user, and continue further computation. If $c > M$, then overflow results. It is usually deemed not worth the added expense to use symmetric rounding and most machines simply chop. Whether a representation is obtained via chopping or symmetric rounding, we shall hereafter refer to the machine representation, $\bar{x}$, as being "rounded." Note that the illustration above was in decimal form for simplicity; analagous results are valid for other bases. For example, if x has the hexadecimal form

$$x = \pm(0.\tilde{a}_1\tilde{a}_2 \ldots \tilde{a}_m\tilde{a}_{m+1}\ldots) \times 16^c, \qquad \tilde{a}_1 \neq 0,$$

then the "decimal" point is really a "hexadecimal" point; and

$$x = \left(\sum_{k=1}^{\infty} (\tilde{a}_k \times 16^{-k})\right) \times 16^c.$$

The chopped form is still obtained by deleting $\tilde{a}_k$, $k \geq m + 1$. Symmetric rounding is done by adding $8 \times 16^{c-m-1}$ to x and then chopping.

Returning to decimal form for simplicity, we first consider the error in symmetric rounding. If $\tilde{a}_{m+1} \geq 5$, then

$$x - \bar{x} = (\pm 0.\tilde{a}_1 \ldots \tilde{a}_m\tilde{a}_{m+1}\ldots) \times 10^c - [(\pm 0.\tilde{a}_1 \ldots \tilde{a}_m) + (\pm 10^{-m})] \times 10^c.$$

If $\tilde{a}_{m+1} < 5$, then

$$x - \bar{x} = (\pm 0.\tilde{a}_1 \ldots \tilde{a}_m\tilde{a}_{m+1}\ldots) \times 10^c - (\pm 0.\tilde{a}_1 \ldots \tilde{a}_m) \times 10^c.$$

In either case $|x - \bar{x}| \leq 0.5 \times 10^{c-m}$, and this is a bound on the absolute error. To get a bound on the relative error, we note that since $\tilde{a}_1 \neq 0$, then $|x| \geq 0.1 \times 10^c$. Thus the relative error satisfies

$$\frac{|x - \bar{x}|}{|x|} \leq \frac{0.5 \times 10^{c-m}}{0.1 \times 10^c} = 5 \times 10^{-m} = 0.5 \times 10^{-m+1}.$$

In a similar manner we can show that the relative error bound from chopping is 10^{-m+1}. It is interesting to note that neither of these relative error bounds depends on the magnitude of x, that is, the size of c. Rather they depend only on the value of m, which is thus said to be the number of "significant digits" of the representation. For example, the IBM System 360 and 370 are hexadecimal with a mantissa of $m = 6$. The relative error from chopping in number representation thus does not exceed 1×16^{-5}. To compare this with the size of relative errors we expect in the decimal mode, we set $1 \times 16^{-5} = 1 \times 10^{-m+1}$. Solving for m yields $m \approx 7$. Thus we have approximately seven-significant-digit decimal accuracy with respect to the relative error of the representation. This statement does *not* mean that any seven-digit decimal number can be represented exactly on this machine as the previous example of $x = 1/10$ shows. The statement says, however, that $|x - \bar{x}|/|x| \leq 10^{-6}$ (approximately, since $m \approx 7$).

In the discussion above, we analyzed the error made by replacing the true value of x by its machine representation $\bar{x}$. Now we wish to assess the effect of each arithmetic operation $(+, -, \cdot, \div)$ to see how errors propagate. Let us use $\bar{x}$ to denote the machine approximation to the true value x where the error, $e(x) = x - \bar{x}$, includes *all* errors that have been made in going from x to $\bar{x}$: that is, $e(x)$ not only includes the error of representation but also includes errors from previous calculations, initial data errors, etc. In other words, $e(x)$ includes all errors from the beginning of the calculation that have led to any discrepancies between x and $\bar{x}$. With $e(y)$ defined similarly for any quantity y, we investigate the error resulting from the "machine addition" of x and y. Now,

$$x + y = (\bar{x} + e(x)) + (\bar{y} + e(y)) = (\bar{x} + \bar{y}) + (e(x) + e(y)).$$

At first glance it seems that the error of the addition is merely the sum of the individual errors. However, this is not always true since even though $\bar{x}$ and $\bar{y}$ can be represented exactly on the machine, it is *not* necessarily true that their sum can also be; i.e., it does not follow that $\bar{x} + \bar{y} = \overline{\bar{x} + \bar{y}}$. For example, consider a four-digit floating-point machine and let $\bar{x} = 0.9621 \times 10^0$ and $\bar{y} = 0.6732 \times 10^0$. Then $\bar{x} + \bar{y} = 1.6353 \times 10^0$. Now it is not uncommon for an m-digit machine to perform arithmetic operations in a $2m$-digit accumulator and then to round the answer. Assuming this situation to be true, we have $\overline{\bar{x} + \bar{y}} = 0.1635 \times 10^1$. Returning to our general analysis, we see that where $z = x + y$, the true error, $z - \bar{z}$, is actually $(e(x) + e(y))$ *plus* the error between $\bar{x} + \bar{y}$ and $\overline{\bar{x} + \bar{y}}$.

Before examining the other three arithmetic operations, let us consider addition somewhat further. We can see immediately that addition can lead to overflow; for instance, $\bar{x} = 0.9621 \times 10^M$ and $\bar{y} = 0.6732 \times 10^M$; thus $\bar{x} + \bar{y}$ does not fall within the range of the computer. Another factor that we must consider is that before a machine can perform an addition, it must align the decimal points. For example, again let $m = 4$ with $\bar{x}_1 = 0.5055 \times 10^4$ and $\bar{x}_2 = \bar{x}_3 = \cdots \bar{x}_1 = 0.4000 \times 10^0$. To perform the addition $\bar{x}_1 + \bar{x}_2$, the computer must shift the decimal four places to the left in $\bar{x}_2$ and form $\bar{x}_1 + \bar{x}_2 = (0.5055 \times 10^4) + (0.0000$

$\times 10^4) = (0.50554 \times 10^4)$, which rounds to $\overline{\bar{x}_1 + \bar{x}_2} = 0.5055 \times 10^4 = \bar{x}_1$. Continuing, we see that

$$\overline{(\cdots(((\bar{x}_1 + \bar{x}_2) + \bar{x}_3) + \bar{x}_4) + \cdots + \bar{x}_{11})} = \bar{x}_1,$$

but

$$\overline{(\cdots(((\bar{x}_{11} + \bar{x}_{10}) + \bar{x}_9) + \bar{x}_8) + \cdots + \bar{x}_1)} = 0.5059 \times 10^4,$$

which is the correct answer. Thus the machine calculates $\Sigma_{i=0}^{10} \bar{x}_{11-i}$ correctly, but not the sum $\Sigma_{i=1}^{11} \bar{x}_i$. This example illustrates the rule of thumb that if we have several numbers of the same sign to add, we should add them in ascending order of magnitude to minimize the propagation of round-off error. The mathematical foundation underlying this statement is that machine addition is not associative; i.e., it can happen that

$$\overline{(\bar{x} + \bar{y}) + \bar{z}} \neq \overline{\bar{x} + (\bar{y} + \bar{z})}.$$

The following numerical examples were run to illustrate this phenomenon. (We used a hexadecimal machine with $m = 6$.) With $x = 1048576 = 16^5 = \bar{x}$ and $y = z = 1/2 = 8/16 = \bar{y} = \bar{z}$, our machine results were

$$(\bar{x} + \bar{y}) + \bar{z} = 0.1048576\text{E}07$$

and

$$\bar{x} + (\bar{y} + \bar{z}) = 0.1048577\text{E}07$$

as our analysis above leads us to expect. Letting $w_0 = 1048576 = 16^5 = \overline{w_0}$ and $w_k = 1/16 = \overline{w}_k$, $1 \leq k \leq 256$, we also obtained the following results (adding in the order indicated by the sum):

$$\sum_{k=0}^{256} w_k = 0.1048576\text{E}07 = w_0$$

but

$$\sum_{k=0}^{256} w_{256-k} = 0.1048592\text{E}07,$$

the latter being the correct result (which we can easily check by hand). As a final example we computed the sum $S \equiv \Sigma_{k=1}^{999} (1/k(k+1)) \equiv 0.999$ by adding forwards and backwards and obtained

$$S \approx 0.9989709\text{E}00 \text{ (forwards)}, \qquad S \approx 0.9989992\text{E}00 \text{ (backwards)}.$$

A concept that is useful in numerical methods is that of *machine epsilon*. Machine epsilon is the smallest positive machine number, ϵ, such that

$$\overline{1 + \epsilon} > 1.$$

(Machine epsilon varies from computer to computer and from compiler to compiler, but a simple program can be written to determine machine epsilon for a given machine and compiler.) As an example to clarify machine epsilon, let us consider a machine (such as the IBM 370) using 6-digit hexadecimal arithmetic. Consider the machine numbers 1 and δ where

$$1 = .100000 \times 16^1 \qquad \delta = .000001 \times 16^1.$$

Clearly the machine addition of $1 + \delta$ will produce $1 + \delta = .100001 \times 16^1$. Therefore $1 + \delta > 1$, and thus machine epsilon is at least as large as $\delta = 16^{-5} = .95367432 \times 10^{-6}$. If the machine arithmetic rounds by chopping, then machine epsilon is exactly equal to δ, for the next smallest machine number is

$$\delta' = .000000\text{F} \times 16^1$$

and (assuming the machine chops) $\overline{1 + \delta'} = 1$.

To see the significance of machine epsilon, consider the following frequently used approximation to $f'(x)$:

$$f'(x) \approx \frac{f(x + h) - f(x)}{h}.$$

In problems in which such an approximation would be used, we would be able to program the machine to evaluate f at any point; and by choosing h smaller and smaller, we would expect to get better and better estimates of $f'(x)$. However if h is small relative to the size of x, then it is quite possible that the machine number x and the machine number $x + h$ are the same. If $x + h = x$ in the machine, then $f(x + h) = f(x)$ in the machine; and we have passed the point at which we can improve our estimate to $f'(x)$ by choosing h smaller. Thus if the approximation is to be reasonable, we need to have (if x and h are machine numbers and $h > 0$)

$$\overline{x + h} > x;$$

or (assuming $x > 0$) we need to have

$$\overline{1 + \frac{h}{x}} > 1.$$

For the approximation to be valid, we need the machine representation for h/x to be at least as large as machine epsilon. In general, machine epsilon gives us the limit to which we can resolve or distinguish two numbers (relative to the size of the numbers).

The error analysis for subtraction is much the same as for addition in that

$$x - y = (\bar{x} - \bar{y}) + (e(x) - e(y)),$$

but now we have the problem that subtraction can cause loss of significant digits. This problem occurs when x and y are nearly equal. For example, if $x = 0.6532849 \times 10^2$, $y = 0.6531212 \times 10^2$, and $m = 4$, then $\bar{x} = 0.6532 \times 10^2$ and $\bar{y} = 0.6531 \times 10^2$ (rounding by chopping). Now $\bar{x} - \bar{y} = 0.1000 \times 10^{-1}$, which is

the machine representation for $\overline{\bar{x} - \bar{y}}$, that is, $\bar{x} - \bar{y} = 0.1000 \times 10^{-1}$. However, $x - y = 0.1637 \times 10^{-1}$. The problem we encounter here is not with this computation itself, but in its effect on later computations since the zeros in $\overline{\bar{x} - \bar{y}}$ have no significance whatsoever and are, in effect, just filling up spaces. The answer is smaller than x or y; and hence errors already present, $e(x)$ and $e(y)$, can have a drastic effect on the relative error of any further calculations since the zeros will be treated thereafter as though they were significant. This source of error is known as *subtractive cancellation* and leads to serious errors in many computations.

When this type of error occurs, we may try to locate the "sensitive" subtraction and eliminate it, if possible, by analytical manipulation. For example, consider the function $f(x) = (1 + x - e^x)/x^2$. Even if we have a very precise way of evaluating e^x, if x is "small," the evaluation of $f(x)$ will suffer from subtractive cancellation in the numerator, which will be magnified by the division by a number very close to zero. One way to help alleviate this problem is to use the Maclaurin series expansion, $e^x = \sum_{k=0}^{\infty} x^k/k! = 1 + x + x^2/2! + x^3/3! + \cdots$. Then $1 + x - e^x = -x^2/2! - x^3/3! - x^4/4! - \cdots$, and we see that the factor $[(1 + x) - (1 + x)]$, which was essentially causing the trouble, is eliminated. Furthermore, $f(x) = -1/2! - x/3! - x^2/4! - \cdots$; thus this analytical technique also eliminates the problem of "division by zero." Since we are considering the evaluation of $f(x)$ for small values of x, this series should be fairly rapidly convergent as evidenced in the following computer results. However, for large values of x, the series is no longer rapidly convergent and we experience considerable difficulties when the series is truncated after a finite number of terms.

We note from the convergent series expansion for $f(x)$ above that $f(0) = -1/2$. In Table 1.2 the series evaluation was truncated after the seventh term; such truncation should theoretically give us at least seven-decimal-place accuracy for the values of x that were used. Table 1.2 demonstrates that the series evaluations yield good answers whereas direct substitution becomes worse as x decreases.

TABLE 1.2

x	$f(x)$ (Direct Substitution)	$f(x)$ (Series Evaluation)
0.1	−0.5171781E 00	−0.5170917E 00
0.01	−0.5054476E 00	−0.5016708E 00
0.001	−0.9536749E 00	−0.5001667E 00
0.0001	−0.9536745E 02	−0.5000166E 00
0.00001	0.0	−0.5000016E 00

Analysis of multiplication yields

$$x \cdot y = (\bar{x} + e(x)) \cdot (\bar{y} + e(y)) = \bar{x}\bar{y} + \bar{x}e(y) + \bar{y}e(x) + e(x)e(y).$$

We can usually assume that the term $e(x)e(y)$ is negligible with respect to the other terms and we omit it. Furthermore, the product of the m-digit numbers, $\bar{x}$ and $\bar{y}$, is at most a $2m$-digit number and can be represented exactly in the $2m$-digit mode that computers commonly use for individual operations. The product will be rounded to m digits, of course, to form its machine representation, $\overline{\bar{x}\bar{y}}$. Thus, even though $\bar{x}\bar{y} \neq \overline{\bar{x}\bar{y}}$ (necessarily), the error is merely that of rounding $\bar{x}\bar{y}$ to its machine representation. Thus the relative error of the difference between $\bar{x}\bar{y}$ and $\overline{\bar{x}\bar{y}}$ is bounded by $0.5 \times 10^{-m+1}$ for symmetric rounding and by $1 \times 10^{-m+1}$ for chopping.

It is common, especially in matrix methods, to have to compute a quantity of the form, $\sum_{j=1}^{n} \bar{x}_j\bar{y}_j$, called an "inner product." The multiplications, $\bar{x}_j\bar{y}_j$, are individually done in the $2m$-digit mode, but often the addtion is not. In that case, it is often a simple task in a high-level language to program the machine to do the additions of this type in double precision as well. This method is called "double precision accumulation of inner products," and serves to lessen the errors resulting from the multiplications, $\bar{x}_j\bar{y}_j$, $1 \leq j \leq n$.

Finally, we analyze the division operation

$$\frac{x}{y} = \frac{\bar{x} + e(x)}{\bar{y} + e(y)} = \left(\frac{x + e(x)}{\bar{y}}\right)\left(\frac{1}{1 + e(y)/\bar{y}}\right).$$

Assuming that $e(y)$ is "small" with respect to $\bar{y}$, we recall the geometric series where for $|r| < 1$

$$\frac{1}{1 + r} = 1 - r + r^2 - r^3 + \cdots.$$

Letting $r \equiv e(y)/\bar{y}$ in the expression above for x/y, we have

$$\frac{x}{y} = \left(\frac{\bar{x} + e(x)}{\bar{y}}\right)(1 - e(y)/\bar{y} + e(y)^2/\bar{y}^2 - e(y)^3/\bar{y}^3 + \cdots) \simeq \frac{\bar{x}}{\bar{y}} + \frac{e(x)}{\bar{y}} - \frac{\bar{x}e(y)}{\bar{y}^2}$$

where we assume the omitted terms in the expansion are negligible with respect to the first three.

Division is much the same as multiplication in that the difference between $\bar{x}/\bar{y}$ and its machine representation $(\overline{\bar{x}/\bar{y}})$ is merely that of rounding the representation for $\bar{x}/\bar{y}$. The preceding expression clearly illustrates the disastrous effect on the total error that division by a value of $\bar{y}$ very close to zero can have. As mentioned earlier, this problem can sometimes be avoided by alternate approaches that analytically circumvent "zero divisions."

PROBLEMS, SECTION 1.3

1. a) Convert the following decimal numbers to hexadecimal form:
   i. 1023 ii. 1025
   iii. 278.5 iv. 14.09375
   v. 0.1240234375
   b) Convert the answers above to binary form.

2. Convert the following hexadecimal numbers to decimal form.
   a) 1023, b) 100, c) 1A4.C, d) 6B.1C, e) FFF.118
3. Convert the numbers π, e, and 1/3 into hexadecimal form with a mantissa of six hexdigits using
   a) symmetric rounding,
   b) chopping.
*4. Prove that any proper fraction that has a terminating hexadecimal expansion also has a terminating decimal expansion. Explain in general terms why the converse of this statement is not valid.
5. For the computer example that computed the sums of reciprocals, compute the relative errors for the operations corresponding to $k = 1000, 1006, 1012, 1018$.
6. a) If x is a real number and $\bar{x}$ is its chopped machine representation on a computer with base 10 and mantissa length m, show that the relative error, $|x - \bar{x}|/|x|$, is bounded by 10^{-m+1}.
   b) If $\delta = 10^{-m+1}$ (in the case of chopping) or $\delta = 0.5 \times 10^{-m+1}$ (in the case of symmetric rounding), show that $\bar{x} = x(1 + \varepsilon)$ where $|\varepsilon| \leq \delta$.
   c) Given a function $F(x)$, F' continuous, we have from the mean-value theorem that in evaluating F at x the relative error from this source alone is

$$\frac{F(\bar{x}) - F(x)}{F(x)} = \frac{F(x + \varepsilon x) - F(x)}{F(x)} = \frac{F'(\xi)[(x + \varepsilon x) - x]}{F(x)} \simeq \varepsilon x \frac{F'(x)}{F(x)}.$$

   Analogously, $F(\bar{x}) - F(x) \simeq \varepsilon x F'(x)$. Use these formulas to give estimates of absolute and relative error if
   i. $F(x) = x^k$, for various values of k and x;
   ii. $F(x) = e^x$, for large values of x;
   iii. $F(x) = \sin(x)$, for small values of x;
   iv. $F(x) = x^2 - 1$, for x near 1;
   v. $F(x) = \cos(x)$, for x near $\pi/2$.
7. Write a program to compute the value of $S_N = \sum_{k=1}^{N} 1/k$ for various values of N. Adapt the program so that it sums forwards and backwards, and compare the results as N increases. Find the value of N such that $S_n = S_N$ for all $n \geq N$ for the forward summation. How is this value reconciled with the fact that theoretically $\lim_{N \to \infty} S_N = \infty$? Would there be such a number N for any modern digital computer and would the number be the same on all such machines?
8. Consider a computer with mantissa length $m = 3$, base $b = 10$, and exponent constraints $\mu = -2 \leq c \leq 2 = M$. How many real numbers will this computer represent exactly?
9. For any positive integer N and a fixed constant, $r \neq 1$, we recall the formula for the geometric sum:

$$G_N \equiv 1 + r + r^2 + \cdots + r^N = (1 - r^{N+1})/(1 - r) \equiv Q_N.$$

   Write a computer program to compute G_N and Q_N for arbitrary values of r and N. Let [illegible] choose to [illegible] both of the [illegible]

check the exact value of Q_N by hand calculation (for example, $r = 1 - 10^{-k}$ for some integer $k \geq 0$). Calculate the relative and the absolute errors of these computations.

1.4 REVIEW OF FUNDAMENTAL MATHEMATICAL RESULTS

In this section we shall briefly list some basic mathematical results, mainly from the calculus, that are useful in the development and investigation of numerical procedures in subsequent chapters. We are, of course, assuming familiarity with the rudiments of calculus such as the real number system, functions and at least an intuitive understanding of the concept of limits (the latter being especially true with respect to sequences since many numerical methods are iterative in nature and thus generate a sequence of approximations to the true solution). A simple example is the classical Newton's method applied to finding the square root of an arbitrary positive real number, α. Given an initial guess x_0 for $\sqrt{\alpha}$, Newton's method generates the sequence: $x_{n+1} = \frac{1}{2}(x_n + \alpha/x_n)$, for $n = 0, 1, 2, \ldots$. Three questions immediately come to mind if we are to be able to use this algorithm successfully in practice. (1) Is the sequence generated by the algorithm convergent to $\sqrt{\alpha}$ and $\sqrt{\alpha}$ alone? (2) How is the convergence related to the choice of the initial guess x_0? (3) How fast does the method converge, or how large is the error, $x_n - \sqrt{\alpha}$, after the nth iteration? These are questions involving the concept of convergence of sequences. There are, of course, other important questions to answer. (4) How does rounding error affect the method? (5) Is it competitive with other methods for finding $\sqrt{\alpha}$? (6) How well can the method be implemented on the particular computer available and what cost is necessary to obtain an acceptable answer?

Other basic ideas necessary from calculus are the concepts of continuity, differentiation, and integration. We have assumed in Chapters 2 and 3 that the reader is able to evaluate determinants. We have, however, tried to keep the material involving determinants minimal, and the other necessary material from matrix theory and linear algebra is self-contained. We have also assumed that the reader is aware of the importance of differential equations in modeling and solving modern technological problems although we assume very little background in the mathematical theory of differential equations in Chapter 7.

We list here some fundamental theorems of calculus and algebra that are used in later chapters and with which the reader should already be familiar. We will present other results only as they are needed to treat particular topics, and the results will be presented only in the context of the particular topic being treated. The following results and their proofs can be found in almost any sophomore-level calculus text.

Theorem 1.1. Rolle's Theorem

If $f(x)$ is a continuous function in the closed interval $[a, b]$ and is differentiable in (a, b), and if $f(a) = f(b)$, then there exists at least one point ξ in (a, b) such