# I. NUMERICAL METHODS

## 1.1. Introduction

One of the principal motivations behind the early development of computers was the need to carry out long numerical calculations quickly and reliably. Computers were soon programmed for other kinds of applications as well, particularly in data processing, but numerical applications continue to be important in such areas as biology, economics and management science, as well as in the physical sciences and engineering.

The distinguishing feature of what we call *numerical* calculations is the presence of numerical errors, such as roundoff or truncation errors, which might accumulate through a long series of numerical operations so that the final result of your calculations may not be very accurate. Some calculations involving numbers (such as determining whether or not a number is prime, or finding greatest common divisors) do not involve numerical errors and are therefore not considered to be *numerical* calculations in this context.

In this chapter we give a brief introduction to numerical methods which can be used to carry out numerical calculations efficiently and with reasonably small errors. By a *numerical method* we mean to include just about everything needed to specify the corresponding computer program: how numbers are represented, how numerical errors occur in a calculation, some useful mathematical formulas for solving numerical problems, rules for getting started, when to stop, etc. However, some further details may still be needed before the corresponding numerical software is completely specified, depending on the particular system (language, machine) being used.

Besides being accurate and efficient in handling the problems for which it was intended, good numerical software will have other desirable characteristics as well, such as being easy to use and reliable. By *reliable* we mean that it will give adequate indications of any difficulties it might have in trying to handle a particular problem, and not just abnormal termination (due to an overflow condition, say) or the return of a wildly inaccurate result.

The mathematical analysis of numerical methods, and in particular of the errors associated with these methods, is usually called *numerical analysis* or *numerical mathematics*.

## 1.2. Number Representation

Most general purpose programming languages include two fundamentally different data types for representing and manipulating numbers. Both representations consist of a string of *digits* and a sign, together with a *point* (decimal point, binary point, etc., whichever is applicable for that particular machine) to indicate the magnitude of the corresponding number.

### 1.2.1. Fixed Point Numbers

In **fixed-point** numbers, the point is understood to be at some *fixed* location within the string. Most commonly, the point is assumed to follow the rightmost digit and the corresponding data type is *integer*. For example, on a decimal machine, the string of digits and the sign could be

+00001234

with the interpretation that the number 1,234 is being represented.

It is also possible to have some other understanding about the location of the decimal point. For example, in dealing with financial calculations, it is convenient to assume that the fixed point is two places in from the right end of the string of digits (between the 2 and the 3 in the example above). Then the number would be interpreted as 12.34, corresponding to 12 dollars and 34 cents.

In most machines, the arithmetic operations carried out by the hardware on fixed point representations treat all numbers as integers (i.e., assume that the decimal point is to the right of the rightmost digit). If a programmer wishes to allow the fixed point to be elsewhere in the interpretation of some or all number representations, then care must be taken. Addition and subtraction must always be done on pairs of numbers with the same placement of the fixed point, and whenever multiplication or division is done, the distance of the fixed point in the result from the right end of the string of digits must be taken to be either the sum or the difference respectively of the distances in the two operands.

### 1.2.2. Floating-Point Numbers

The other way of representing and manipulating numbers is called **floating-point**. Here the representation corresponds to a *scientific notation* where each number has a separate exponent part, as in

$$1.234 \times 10^5, \quad .678 \times 10^{-7}, \quad -307.0 \times 10^{10}.$$

The terminology suggests that the position of the *point* could ''float'' from one position to another, according to the value stored in the *exponent.* We usually say that numbers represented in floating-point form have the data type *real* although actually only a small subset of the real numbers can be exactly represented in this form.

A basic unit of storage in a computer is a *word*. Typically, a *word* holds an integer, a rational number, a few characters, or a machine instruction. To illustrate the *representation* of floating-point numbers, suppose a *word* consists of a sign and six decimal digits. Four of these digits can be used to represent the four most significant digits of the number, and the other two can be used to represent the exponent (including its sign, as we shall explain below). We will display the internal representation of a floating-point number as $\pm\,.FFFFe\pm XX$ where $FFFF$ are the four most significant digits of the number and $XX$ are the two decimal digits of the base 10 exponent. (Most machines use base 2 or base 16 exponents, rather than base 10, but it is easier to discuss examples using the more familiar base 10.)

Below we consider only *normalized* representations. A representation is said to be normalized if its decimal point is immediately to the left of its most significant digit, thus $.1234 \times 10^5$ is the normalized form of both $1.234 \times 10^4$ and $.001234 \times 10^7$. The fraction part of a number that has been normalized in the way described here is sometimes called the *significand*. It is also sometimes called the *mantissa*, by analogy with the meaning of the term in connection with logarithms. With our chosen convention for displaying representations,

| | | | | | |
|---|---|---|---|---|---|
| $1.234 \times 10^4$ | is | $+.1234e+05$, | $12$ | is | $+.1200e+02$, |
| $-7.8$ | is | $-.7800e+01$, | $3.4 \times 10^{-6}$ | is | $+.3400e-05$, |
| $.003$ | is | $+.3000e-02$, | $1.0 \times 10^{20}$ | is | $+.1000e+21$. |

One motivation for using a normalized representation is to make sure each real number, other than zero, has a unique representation. Without normalization, the number +12 could be represented as $+.1200e+02$ or $+.0120e+03$ or $+.0012e+04$. Because the number zero contains no non-zero significand digits, it requires a special normalization convention. We will represent it as $+.0000e+00$.

In our hypothetical decimal representation consisting of a sign and six decimal digits, the sign bit can represent the sign of the significand, but we are left with the problem of representing the sign of the exponent even though a second explicit sign bit is not available. One way in which this can be done is by using *excess* notation for the exponent part. For example, in the case treated here, we have two decimal digits to represent the exponent, which gives us 100 distinct representations ("00" to "99"). In order to allow negative as well as positive exponents without requiring an explicit sign bit, we could use "excess 50" notation, which requires that an exponent be increased by 50 to determine its internal representation. Thus, an exponent of +2 is encoded as "52" internally. With this approach, the representations "00" to "99" correspond to exponents ranging from –50 to +49.

#### 1.2.2.1. IEEE Standard Floating-Point Numbers

A standard for floating-point arithmetic was approved in 1985. It is known as the IEEE standard, because it was developed by the Institute of Electrical and Electronic Engineers. (The result was primarily due to the efforts of one person, W. Kahan, a graduate and former faculty member of the U of T, now at UC Berkeley.)

The IEEE standard provides specifications for words of 32 or 64 binary digits, or bits. With 32 bits, the first ($s$) represents the sign, the next 8 ($e$) represent the exponent of the base 2, plus 127, and the remaining 23 ($f$) represent the significand without the most significant bit (which must be 1 for non-zero numbers); and $e$ must satisfy $0 < e < 255$. Then the number represented is $(-1)^s 2^{e-127}(1.f)$.

Notice that the normalization convention is slightly different from the decimal example we introduced earlier. Notice also that, because $e$ is restricted, there are still many bit patterns that have not yet been specified. If both $e$ and $f$ are zero, then the numbers represented are $(-1)^s 0$, i.e., +0 and –0. The remaining bit patterns are used for a variety of other purposes.

One of the most important requirements of the IEEE standard is that the default results of all arithmetic operations $(+, -, *, /)$ must be properly rounded, provided of course that they do not overflow or underflow. This requirement has forced almost all manufacturers to build reliable arithmetic units, rather than the variety of sloppy constructions which plagued the industry for so many years.

### 1.2.3. Exercises

1. What is the largest number that can be represented in our hypothetical floating point form with 4 significant digits? What is the smallest positive number that can be so represented? What, roughly, are the corresponding numbers for the 32-bit IEEE format?

## 1.3. An Example Calculation: How Not to Sum a Series

A problem that sometimes arises in numerical calculations is to find an approximation to the sum of an infinite series. Quite often the series is what's called a Taylor series, such as the following example:

$$e^x = 1 + x + x^2/2! + x^3/3! + \cdots + x^n/n! + \cdots .$$

Since the function $n!$ grows so much more quickly than does $x^n$ for large $n$, it can be shown that this series converges for all values of $x$. Thus, from a strictly mathematical point of view, we have ''solved'' the problem of estimating $e^x$ for any particular value of $x$, since an approximation to $e^x$ can be constructed, which is as accurate as we wish, by summing sufficiently many terms in the series. However, this is in general not practical from a computational point of view. First, many terms are needed when $x$ is large. And second, because some [additional] numerical error will in general be introduced at each step in the calculation, *even if we were to sum up the required number of terms,* the resulting estimate might still be highly inaccurate.

For example, consider the problem of calculating $e^{-10} \approx 4.53999 \times 10^{-5}$ to 6 decimal places of accuracy. In this case, the series above becomes

$$e^{-10} = 1 - 10 + 100/2 - 1000/6 + 10^4/24 - 10^6/120 + \cdots$$

Notice that, although we have a convergent alternating series, the magnitudes of the first 10 terms are *increasing* (reaching a maximum of about 2755.73 for $n = 9, 10$) before the increase in magnitude in the denominator, i.e., $n!$ versus $(n-1)!$, finally exceeds the increase in magnitude in the numerator, namely $10^n$ versus $10^{n-1}$. Thus, for our calculations to produce an accurate result we must include a large number of terms from the summation, which increases its running time. Indeed, after trying various values of $n$, we find that at least 47 terms must be included in the summation in order to obtain six decimal places of accuracy.

However, there is a more serious problem with this calculation than just its large running time: if these calculations are done using limited precision floating point arithmetic (as opposed to ''ideal'' real arithmetic) then the computed result may contain a significant amount of numerical error. (Remember that the 6th significant figure in the final answer is actually the 9th digit to the right of the decimal place, and thus corresponds to the 13th significant figure in the terms for $n = 9, 10$.) Indeed, as shown in Figure 1, it may be impossible to compute a reasonable approximation to $e^x$ for $x$ negative and much less than zero using this method. In particular, when each floating point value in the computation is rounded to 6 decimal places, we observe an error of about one unit in the *fourth* place for $x = -5$ and about one unit in the *third* place for $x = -6$. By $x = -9$, the error has reached the level of one unit in the *most significant* place, and for $x \le -10$ the computed result just seems to be swinging wildly about. The corresponding results when each floating point value in the computation is rounded to 9 decimal places are slightly better, but even here the result has the wrong decimal order of magnitude for $x \le -13$, and the wild oscillations are again evident for $x \le -17$. Obviously, we have some work to do before we can come up with a *reliable* function for calculating $e^x$. (We return to this question in Subsection 1.4.4.)

## 1.4. Sources of Numerical Error

For purposes of illustration, let us continue to assume that floating point numbers are represented using a sign and 6 decimal digits, of which the sign and 4 of the digits are used to represent the significand with the remaining 2 digits representing the exponent in excess 50 notation. In this case, we can identify four distinct sources of numerical error in a calculation such as the example above.

| $x$ | $e^x$ | floating point approximation using... | |
|---|---|---|---|
| | | 6 decimal places | 9 decimal places |
| –1 | 0.367879 | 0.367879 | 0.367879 |
| –2 | 0.135335 | 0.135336 | 0.135335 |
| –3 | 0.049787 | 0.049787 | 0.049787 |
| –4 | 0.018316 | 0.018316 | 0.018316 |
| –5 | 0.006738 | 0.006737 | 0.006738 |
| –6 | 0.002479 | 0.002489 | 0.002479 |
| –7 | 9.11882e–4 | 9.829029e–4 | 9.11907e–4 |
| –8 | 3.354626e–4 | 4.121081e–4 | 3.356439e–4 |
| –9 | 1.234098e–4 | 9.150566e–4 | 1.238699e–4 |
| –10 | 4.539993e–5 | –9.702667e–4 | 4.513337e–5 |
| –11 | 1.67017e–5 | –0.002845 | 2.195272e–5 |
| –12 | 6.144212e–6 | 0.013705 | 5.919018e–6 |
| –13 | 2.260329e–6 | –0.032359 | 4.587181e–5 |
| –14 | 8.315287e–7 | 0.03735 | 3.499875e–5 |
| –15 | 3.059023e–7 | 0.160922 | 1.491768e–4 |
| –16 | 1.125352e–7 | –0.107055 | 7.689512e–4 |
| –17 | 4.139938e–8 | 2.506252 | –1.552075e–4 |
| –18 | 1.522998e–8 | –2.729362 | 0.001737 |
| –19 | 5.602796e–9 | 11.741333 | –0.002737 |
| –20 | 2.061154e–9 | 8.911186 | 0.01936 |
| –21 | 7.58256e–10 | –3.439907 | –0.04547 |
| –22 | 2.789468e–10 | –316.803223 | –0.248811 |
| –23 | 1.026188e–10 | 4.391659 | –0.521117 |
| –24 | 3.775135e–11 | –1173.568359 | 3.680798 |
| –25 | 1.388794e–11 | –4242.597656 | –4.717535 |
| –26 | 5.109089e–12 | –11509.328125 | 5.659895 |
| –27 | 1.879529e–12 | –15150.226562 | 46.790298 |
| –28 | 6.9144e–13 | 84113.5625 | 60.710347 |
| –29 | 2.543666e–13 | –336932.25 | 51.44295 |
| –30 | 9.357623e–14 | 119399.6875 | –1481.176309 |

Figure 1. Approximate calculations of $e^x$.

### 1.4.1. Representation Error

Any number that has more than four significant digits must be shortened in some way before it can be represented in a machine word as described above. For example, 13,579 is represented as either $+.1358e+05$ or $+.1357e+05$, depending on whether *rounding* or *chopping* is used to reduce the number of significant digits to four. Similarly, ⅓ would be represented as $+.3333e+00$ with either rounding or chopping, while ⅔ would be represented as $+.6666e+00$ with chopping but as $+.6667e+00$ with rounding. A third way of shortening the number is called *proper rounding*. This involves applying a special rule whenever the $d+1^{st}$ digit is $b/2$ for base $b$ (i.e., 5 in base 10) that requires that the shortening be done such that the last digit is made *even*. Thus .16805 would round down to .1680, whereas .16815 would round up to .1682, while .99995 and 1.0005 would both round to 1.000. The advantage of this rule is that the rounding is *unbiased* (which means that we expect to round up and round down with nearly equal frequency over a period of time).

The difference between an arbitrary real number and the $d$-significant-digit approximation by which it is represented is called the **representation error.**

### 1.4.2. Roundoff Error

In addition to representation error, **roundoff error** can arise whenever arithmetic operations are performed on floating-point numbers. Roundoff error can occur with any of the four arithmetic operations, even if we were lucky enough to have all of the required operands exactly representable in our floating point number system. For example, with the above representation, adding .1234 (+.1234*e*+00) to .04567 (+.4567*e*−01) has the exact result .16907, which cannot be represented with only four significant digits. As in representing numbers initially, any of chopping or rounding or proper rounding can be used to shorten the number in order to make it representable. In the example, chopping would give .1690 (+.1690*e*+00), while rounding (and proper rounding) would give .1691 (+.1691*e*+00). Note that some roundoff error is incurred no matter which of the three options is taken.

To determine the result of a particular arithmetic operation it may be best to work with the numbers in unnormalized form. For example, suppose +.1234*e*−01 and +.5678*e*−02 are the floating-point representations of two numbers and we wish to find the floating-point representation of their sum. The two numbers are .01234 and .005678, and their exact sum is .018018, which contains more than four significant digits. Their properly rounded sum is .01802, whose floating-point representation is +.1802*e*−01, and their chopped sum is .01801, whose floating-point representation is +.1801*e*−01.

Floating-point arithmetic is *commutative*, but simple examples can easily be constructed to show that it is neither *associative* nor *distributive*. (*Commutative* means that $a \times b = b \times a$ and $a + b = b + a$. The other terms are defined in exercise 2 at the end of this section.)

If a result is too large in magnitude to be represented in the machine (an exponent > +49 in the excess 50 representation considered above), it is said to *overflow*. If too small in magnitude (an exponent < −50), it said to *underflow*. Machines usually *trap* on overflow or underflow (i.e., interrupt execution and signal an error condition), and it is left up to the language and system designers to determine what can happen when such conditions occur. Unfortunately, there are no standards to follow, and the handling of overflow and underflow can sometimes be quite frustrating in practice. On the other hand, these conditions do not arise very often. Usually, overflow means that there is an error in the program. On the other hand, when underflow occurs, it is often appropriate to simply replace the number by zero. Machines will generally trap on *zero divide* as well as on overflow or underflow.

### 1.4.3. Cancellation and the Quadratic Formula

*Cancellation* occurs when nearly equal numbers are subtracted (or equivalently, when numbers of equal magnitude but opposite sign are added). Cancellation can be a major worry with floating-point arithmetic, although cancellation does not, by itself, introduce any new error in the results. For example, in the four-digit arithmetic introduced above, .9999–.9998 = .0001, which can be represented exactly. Note, however, that each of the numbers on the left side of the equation consist of four significant digits, and that the subtraction turns three of those four into zeroes. When .0001 is normalized and represented as +.1000*e*−03, any error in the *last* significant digit becomes an error in the *first* significant digit. While an error of half in the *last* place of .9999 may be considered negligible in comparison to .9999, an error of half in the *first* significant digit of .1000 is clearly not negligible. The *relative error* changes from .005% before the subtraction to about 50% after the subtraction, an increase by a factor of 10,000.

One example where this type of cancellation is important is in solving quadratic equations. If we use the usual formulas,

$$x_1 = (-b + \sqrt{b^2 - 4ac}\,)/2a$$
$$x_2 = (-b - \sqrt{b^2 - 4ac}\,)/2a$$

to solve the equation

$$a\,x^2 + b\,x + c = 0,$$

there is a danger that cancellation will cause one of the computed results to have a large relative error.

To illustrate this point, consider the following program.

```
var a, b, c, x1, x2 : real
get a, b, c
x1 := (– b + sqrt (b ∗ b – 4 ∗ a ∗ c)) / (2 ∗ a)
x2 := (– b – sqrt (b ∗ b – 4 ∗ a ∗ c)) / (2 ∗ a)
put x1, " ", x2
```

This program will quite often work satisfactorily. For example, if the calculations are done in precision 4, properly rounded, input of `1, 3, 1` produces output of `-0.3820  -2.618,` where the true roots, rounded to precision four, are –0.3820 and –2.618. However, input of `1, 99, 1` produces output of `-.01000 -99.00,` whereas the true roots, rounded to precision four, are –0.01010, –98.99. Here the relative error in the first root is 100 times that in the second root! This is caused by cancellation between $-b$ and $\sqrt{b^2 - 4ac}$ in the calculation of the first root.

One way out of this difficulty is to use only the computed result for the one root that does *not* suffer from cancellation. This means choosing $-b + \sqrt{b^2 - 4ac}$ or $-b - \sqrt{b^2 - 4ac}$, whichever has no cancellation, to compute that one root. The other root can then be obtained from the fact that the product of the roots must be $c/a$. That is, it can be obtained by dividing the first root into $c/a$. The following program accomplishes this task:

```
var a, b, c, x1, x2 : real
get a, b, c
if b < 0 then
   x1 := (– b + sqrt (b ∗ b – 4 ∗ a ∗ c)) / (2 ∗ a)
else
   x1 := (– b – sqrt (b ∗ b – 4 ∗ a ∗ c)) / (2 ∗ a)
end if
x2 := c / (a ∗ x1)
put x1, " ", x2
```

Here input of `1, 3, 1` produces output of `-0.3820   -2.618,` as before, and input of `1, 99, 1` produces output of `-0.01010 -99.00.` In both cases the results are now satisfactorily accurate.

Another way of avoiding the cancellation problems in some situations is to do the calculations in higher precision. (This is how the "true" values quoted above were obtained.) However, this approach is not usually available in standard programming languages.

Of course, a complete program for solving quadratic equations would have to contend with other difficulties besides possible cancellation. Provision would have to be made for the possibility of complex roots, and for possible exceptions such as overflow, underflow or zerodivide (see exercise 9).

Another situation in which cancellation can be important is in finding sums of series. Indeed, cancellation was the major contributor to the numerical difficulties we encountered with the exponential function earlier in this chapter. A simple trick for avoiding the difficulty with this particular series is to evaluate $1/(e^x)$ when $e^{(-x)}$ is required, so that there is no cancellation at all. Unfortunately, a trick such as this is only available in very special situations. It would be nice to have more general techniques, such as higher precision, to fall back on when necessary.

### 1.4.4. Truncation Error and the Exponential Function

**Truncation error** occurs whenever we do not carry out all the required steps in a mathematical calculation, for example, in finding the sum of an infinite series or the fixed point in an iteration formula. In some cases, such as a truncated Taylor series, where we use only a finite number of terms from the series, general formulas for the truncation error are known. For example, in the case of the exponential series described above, we know that

$$e^x = 1 + x + x^2/2! + \cdots + x^n/n! + R_n ,$$

where the remainder (or truncation error), $R_n$, is given by

$$R_n = e^{\xi} x^{n+1}/(n+1)! ,$$

and $\xi$ is some [unknown] number in the interval between 0 and $x$. The formula is true in general, but we will use it[1] only for $0 \le x \le 1$, in which case the error in $e^x$ is bounded by $e/(n+1)!$ Trying a few values of $n$, we soon find that

[1] Because of the numerical problems that can arise in summing the above series for arbitrary values of $x$, we will now restrict ourselves to the special case when $0 \le x \le 1$. Our objective will be to study the calculation in some detail, to demonstrate that we can obtain about 6 decimal places of accuracy in the approximation, even though we assume that all floating-point computations are carried out using properly rounded 6-digit normalized decimal arithmetic. Afterwards, we will describe how to obtain answers to the general case in terms of this special case and a few simple range reduction rules.

$n = 9$ yields a bound of less than $10^{-6}$, i.e., less than .1 unit in the last place (.1 ulp). It is .1 unit in the last place because the value we are approximating is in the interval [1, $e = 2.732...$] and, for 6-digit numbers in this interval, a unit in the last place means a unit in the 5th place after the decimal point. As far as truncation error is concerned, the series up to and including the term $x^9/9!$ is therefore accurate enough for our purposes.

The most efficient way of calculating the value of the remaining polynomial

$$1 + x + x^2/2! + \cdots + x^9/9!$$

is to rewrite it according to what is called Horner's rule, which produces

$$((((((((1/9!)x + 1/8!)x + 1/7!)x + 1/6!)x + 1/5!)x + 1/4!)x + 1/3!)x + 1/2!)x + 1)x + 1$$

and to do the calculations in the order suggested by this form. The constants 1/9!, 1/8!, etc., can be precomputed, and then only 9 multiplies and 9 adds are needed to evaluate the polynomial in this form.

```
function easyexp (x : real) : real
   % returns exp(x) within 1.3 ulps for 0 <= x <= 1.
   pre 0.0 <= x and x <= 1.0
   if x < 1e-3 then
       result 1.0 + x
   else
       result ((((((((2.75573e-6 * x + 2.48016e-5) * x + 1.98413e-4) * x + 1.38889e-3) *
          x + 8.33333e-3) * x + 4.16667e-2) * x + 1.66667e-1) * x + 5.00000e-1) *
          x + 1.00000) * x + 1.00000
   end if
end easyexp
```

Figure 2. A function for determining an approximation to $e^x$ within less than 1.3 units in the last place for any $x$ in $[0,1]$. It is assumed that all calculations are done in properly rounded, 6-digit, normalized, decimal, floating-point arithmetic.

For the function in Figure 2, the coefficients have been computed and properly rounded to 6 significant decimal digits. The function uses the resulting expression, except that it uses the simple form $1 + x$ when $x < 10^{-3}$. Using only two terms in this latter case is of course much more efficient than using all ten terms in the general case. But this is not the reason for using this shorter form. (In fact, the cost of the test is greater than any saving, on average, from occasionally being able to use only two terms.) The real reason is to avoid the underflow that would occur with extremely small values of $x$ in the product of $2.75573 \times 10^{-6}$ and $x$.

The truncation error with the simpler form is bounded by $e^{.001}(.001)^2/2 \approx 0.5 \times 10^{-6}$. There will in general also be a roundoff error of at most $.5 \times 10^{-5}$ in forming the sum. The total error in the final result is therefore less than $.6 \times 10^{-5}$, or .6 units in the last place (ulps).

As already indicated, the truncation error with the more complicated form is less than $10^{-6}$. The evaluation of the polynomial also involves a number of roundoff errors, but we will see that the net effect of these errors, along with the errors in the computed values of the coefficients, will amount to less than 1.2 ulps. Combined with the truncation error of .1 ulps, this means a total error of less than 1.3 ulps.

To see that the polynomial is evaluated to within an error of less than 1.2 ulps, we have to examine carefully the individual steps in the calculation. First, consider the evaluation of

$$2.75573e{-6} \times x + 2.48016e{-5}$$

The first coefficient, rounded to 8 places, is $2.7557319e{-6}$. The error in the first coefficient, after multiplication by $x$, therefore contributes less than .2 ulps to the error in the product, which is .02 ulps of the expression. The multiplication itself contributes at most .5 ulps to the product, which is .05 ulps of the expression. The addition can contribute up to .5 ulps. And, finally, the error in $2.48016e{-5}$ is less than .14 ulps, since its more accurate value, rounded to 8 places, is $2.4801587e{-5}$. The total of these four bounds is .71 ulps.

This rather tedious argument can be continued, with obvious modifications, through each stage in the evaluation of the polynomial. The error bound in the final result turns out to be less than 1.2 ulps, as stated earlier, to establish the overall error bound of 1.3 ulps for *easyexp*.

We now consider what can be done to handle values of $x$ which are not in the interval $[0,1]$. For example, if $1 < x \le 2$, we could compute an approximation to $e^{x/2}$, and then square the result. Since $(e^{x/2})^2 = e^x$, this would give an approximation to $e^x$ in the new range. More generally, for any $x > 1$, we could determine $k$ such that $2^{k-1} < x \le 2^k$, then approximate $e^{x/2^k}$, and then square this approximation $k$ times. Some provision should be made for rejecting values of $x$ which are so large that $e^x$ would overflow. Values of $x < 0$ can be handled by computing an approximation to $e^{|x|}$, and then taking the reciprocal.

The further $x$ is from the range $[0,1]$, the greater will be the error bound. Some use of higher precision during portions of the calculations is needed to overcome this difficulty. This is not easy to achieve in most programming languages, and this is one of the reasons why functions such as the exponential function (and sin, cos, log, etc.) are usually implemented in what appear to be mysterious ways in assembler language, and are also often not very reliable.

One final comment should be made about the polynomial approximation provided by a Taylor's series. Such series tend to be better approximations near one end of the interval in which they are used (near the origin in the case of the exponential series discussed in this section). Better approximations are available, sometimes in polynomial form but also sometimes in rational form. They are better in the sense that they approximate the desired function more uniformly across the interval of interest, and they are able to do so with smaller numbers of terms, and hence with fewer arithmetic operations required for their evaluation. Such approximations are *minimax* (the maximum error in the interval is a minimum). They are sometimes called Chebyshev approximations, after the mathematician who first wrote extensively about such approximations more than 130 years ago.

### 1.4.5. Exercises

1. In each of the following, the floating-point representations of two numbers are given along with an arithmetic operator. Find the floating-point representation of the corresponding results obtained with properly rounded arithmetic and with chopped arithmetic.

   (a) $+.4321e{+}03$ $+.8765e{+}01$ $+$

   (b) $+.4321e{+}03$ $+.8765e{+}01$ $-$

   (c) $+.8000e{-}01$ $+.3000e{-}11$ $*$

   (d) $+.8000e{-}01$ $+.3000e{-}11$ $/$

   (e) $+.1234e{+}00$ $-.1234e{-}02$ $+$

   (f) $+.1234e{+}00$ $-.1234e{-}02$ $-$

   (g) $+.1000e{+}10$ $+.1000e{+}07$ $+$

   (h) $+.1000e{+}10$ $+.1000e{+}07$ $-$

   (i) $+.1000e{+}10$ $+.1000e{+}06$ $+$

   (j) $+.1000e{+}10$ $+.1000e{+}06$ $-$

   (k) $+.1001e{+}00$ $+.1001e{+}00$ $*$

2. Give examples of four-digit numbers to show that floating-point arithmetic is neither distributive ($a \times (b + c) = a \times b + a \times c$) nor associative ($(a + b) + c = a + (b + c)$ and $(a \times b) \times c = a \times (b \times c)$), regardless of whether proper rounding or chopping is used.

3. Most machines do not obey the simple rules for floating-point arithmetic that we have described. (To give just one example of how carelessly designed some arithmetic units can be, there is a well-known machine that *may* produce the value *false* for the logical expression $1.0 \times a{=}a$.) But even with our simple rules there can be some surprises. Assuming a decimal representation with a two place exponent and a four place fraction, find two numbers, $a$ and $b$, so that the value produced for the expression $(a + b)/2$ is *outside* the closed interval $[a,b]$.

4. In each of the following, the floating-point representations of two numbers are given along with an operator. What happens in each case when an attempt is made to evaluate the corresponding expression, in either

properly rounded or chopping arithmetic?

(a) $+.2222e{+}60 \quad +.2222e{+}60 \quad *$

(b) $+.5000e{+}99 \quad +.5000e{+}99 \quad +$

(c) $+.4000e{+}99 \quad +.6000e{+}99 \quad -$

(d) $+.2222e{+}60 \quad +.2222e{-}60 \quad /$

(e) $+.3333e{-}60 \quad +.3333e{-}60 \quad *$

(f) $+.1111e{-}98 \quad +.1110e{-}98 \quad -$

(g) $+.1000e{+}00 \quad +.0000e{+}00 \quad /$

5. With three-digit representations of the fraction part instead of four-digit, what happens in each of the following cases:

(a) $+.110e{+}01 \quad +.909e{+}99 \quad *$

(b) $+.110e{+}00 \quad +.909e{-}99 \quad *$

6. Suppose you want to approximate the sum of a series in which the terms are steadily decreasing in magnitude. Here is an example:

$$1 + 1/2^3 + 1/3^3 + 1/4^3 + \cdots + 1/100^3$$

Is the result likely to be more accurate if the sum is done from left to right, or from right to left? Why?

7. It can be shown mathematically that the truncation error in approximating the sum of a series where terms decrease in magnitude and alternate in sign is bounded by the magnitude of the first neglected term. How many terms need to be used from the Bessel series

$$J_0(x) = 1 - x^2/(2!)^2 + x^4/(4!)^2 - x^6/(6!)^2 + ...$$

to ensure a truncation error for $-2 \le x \le 2$ of no more than $10^{-6}$?

8. Extend the *easyexp* program to provide an approximation to $e^x$ for all $|x| < 100$. How would you test such a program if it were running on a 6 decimal digit floating-point machine, and no built-in elementary functions were provided?

9. Develop a procedure for solving quadratic equations. The procedure should expect to be provided with the real coefficients $a$, $b$, $c$ and it should return an integer indicator, say *ind*, along with, if possible, two reals, $x1$ and $x2$. The value of *ind* should indicate what type of situation has occurred (two real roots, one real root, complex conjugate roots, no roots, otherwise(?)) and the value of $x1$, $x2$ should define the roots, if any. (For example, $x1$ could be the real part and $x2$ the imaginary part in case the roots are complex.) You should be able to look after all possible eventualities quite easily, except for overflow or underflow. Overflow and underflow can be handled, but it is usually a very messy task, and is not expected in the procedure asked for in this question.

### 1.5. Evaluating Elementary Functions

The discussion so far in this chapter has been concerned mostly with arithmetic operations. Floating-point calculations also involve the elementary functions (including *sqrt*, *sin*, *cos*, *ln*, *exp* and *arctan*), along with a number of other built-in functions such as *div* and *mod*.

In this section, we show of how two of the elementary functions, *sqrt* and *sin*, can be evaluated in terms of only the arithmetic operations and a few closely related built-in functions. We will name the functions *squareroot* and *sine*, instead of *sqrt* and *sin*, because the latter two names are usually reserved words. Our purpose is to show the kind of calculations that are carried out when elementary functions are involved, and to thereby help a potential user understand what can be expected of these functions in terms of accuracy, efficiency and argument limitations.

### 1.5.1. Fixed-Point Iteration and Square Root

Infinite series, such as the one we used for $e^x$, are used for evaluating all of the elementary functions except *sqrt*. For *sqrt* we use another method known as **fixed-point iteration.** The idea here is to solve an equation of the type:

$$x = f(x).$$

One approach to solving this type of equation is to start with an initial guess $x_0$, plug it into the right-hand side of the equation to get a new approximation $x_1$, and repeat until the difference between two successive approximations, say $x_n$ and $x_{n+1}$, becomes very small (in which case we assume that the method has *converged* to a *fixed-point* for the iteration formula, which is approximately $(x_n + x_{n+1})/2$) or a limit to the maximum number of iterations is reached (in which case we assume that the method failed to converge).

Heron's formula, an ancient method for computing the square root of any positive number $a$, is a well known application of fixed-point iteration. It is equivalent to applying the fixed-point iteration technique to the equation $x^2 - a = 0$, after first rewriting this equation in the form $x = .5(x + a/x)$. (Other forms are possible, but this one is quite simple and happens to converge very nicely.) Starting with a positive value $x_0$, a sequence of approximations $x_1, x_2, x_3, \ldots$ is then generated using the formula:

$$x_{n+1} = .5 \times (x_n + a/x_n),$$

for which it can be proven that $x_n \to \sqrt{a}$ as $n \to \infty$. Furthermore, the rate of convergence is very rapid, so that only a small number of iterations is required, provided a reasonably good starting approximation $x_0$ can be obtained. (See exercise 1 at the end of this section.)

Normally, in writing a square root routine for one's own use, it is not necessary to work hard to get a good initial estimate, since convergence always occurs from any starting value. The simple function below is based on this approach.

```
function squareroot (x : real) : real
   % simple (not-so-efficient) squareroot
   pre x >= 0
   var approx : real := 1
   loop
        approx := 0.5 * (approx + x / approx)          % Heron's formula
        exit when abs (approx – x / approx) < .0000001
   end loop
   result approx
end squareroot
```

Figure 3. A simple function for computing square roots.

However, in writing a square root routine for inclusion in a program library (which is to be used very frequently by a large number of users), it is worthwhile to strive for the greatest possible efficiency. In the remainder of this subsection, we describe a more efficient implementation of the square root function, which is shown below in Figure 4. This function uses a very craftily constructed initial estimate in order to guarantee that three iterations of the calculation will suffice to yield sufficient accuracy in 8-digit arithmetic.

The function is able to obtain a value of $x_0$ that has about 1 decimal place of accuracy. Each iteration approximately doubles the number of decimal places of accuracy, so that only three iterations are needed to obtain approximately eight decimal places of accuracy.

For the purposes of our illustration, we will assume that properly rounded eight-digit arithmetic is used throughout.

To describe the program in more detail, we note that the argument of the function satisfies either $x = 0$ or $x = \pm f \times 10^e$, where $f$ is the normalized fraction ($.1 \le f < 1$) and $e$ is the integer exponent.

The squareroot function returns an error condition if $x < 0$ (line 4), or the value 0 if $x = 0$ (lines 5-7). Otherwise it returns a good approximation to

$$\sqrt{x} = \begin{cases} \sqrt{f} \times 10^{e/2}, & \text{if } e \text{ is even,} \\ \sqrt{f}/\sqrt{10} \times 10^{(e+1)/2}, & \text{if } e \text{ is odd.} \end{cases}$$

```
1   function squareroot (x : real) : real
2     % squareroot, assuming the base is decimal;
3     % the result is within 4 ulps if the precision is <= 8
4     pre x >= 0
5     if x = 0 then
6         result 0
7     end if
8     var f := setexp (x, 0)                % fraction part of x
9     var e := getexp (x)                   % exponent part of x
10    var approx := 0.2591 + 0.8194 * f     % rough approx to root of f
11    if e mod 2 not= 0 then
12        e := e + 1                        % make adjustments
13        approx := approx / 3.162          % for odd exponent
14    end if
15    approx := setexp (approx,             % multiply by 10**(e/2)
16        getexp (approx) + e div 2)        % to get approx to root of x
17    for i : 1 .. 3
18        approx := 0.5 * (approx + x / approx)        % Heron's formula
19    end for
20    result approx
21  end squareroot
```

Figure 4. A more efficient function for computing square roots.

When $e$ is even, the answer is simply $\sqrt{f}$ as the fraction, and $e/2$ as the exponent. When $e$ is odd, the exponent of the result (which must be an integer) is taken as $(e+1)/2$, so it is necessary to divide $\sqrt{f}$ by $\sqrt{10}$ to form the fraction part. To obtain this approximation, the program first determines $f$ and $e$ (lines 8-9). Then it computes (line 10) an initial approximation to $\sqrt{f}$ from the expression

$$.2591 + .8194 \times f$$

It can be seen from the graph below that this is a fairly good approximation to $\sqrt{f}$ in the interval [.1,1). (It can be shown mathematically that, except for minor rounding errors in the coefficients, it is the best possible linear approximation to $\sqrt{f}$ in this interval, in the sense that the maximum relative error is a minimum.)

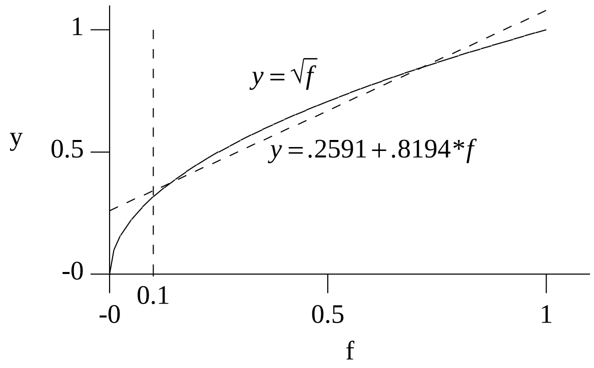



Figure 5: The initial approximation to $\sqrt{f}$.

When $e$ is odd, the exponent is increased by 1 and the approximation to $\sqrt{f}$ is divided by 3.l62, which is approximately $\sqrt{10}$ (lines 11-14). The fraction and exponent parts are then combined to produce the initial approximation to $\sqrt{x}$ (lines 15-16). This approximation is $a_0$, and it can be shown that its relative error is less than $10^{-1}$. Heron's formula is then applied three times (lines 17-20). It can be shown that the final result has a relative error of at most four ulps (four units in the last place) when all calculations are done in properly rounded eight-digit arithmetic. Very little of this error is due to Heron's formula. Most of it is caused by roundoff errors that occur in the last iteration.

### 1.5.2. Range Reduction and the Sine function

Approximations to sin($x$) can be based on the Taylor series

$$\sin(x) = S_n(x) + E_n(x),$$

where $S_n(x)$ is the polynomial

$$S_n(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots - \frac{(-1)^n x^{2n-1}}{(2n-1)!}.$$

The *truncation error*, $E_n(x)$, satisfies

$$|\, E_n(x)\, | \le \frac{|\, x\, |^{2n+1}}{(2n+1)!}\ , \quad \text{provided } x^2 \le (2n+2)\times(2n+3)$$

For any given $x$, this error term can be made as small as we like by using a sufficiently large value of $n$. If an appropriate value of $n$ is known then we can approximate sin($x$) simply with the polynomial $S_n(x)$.

There are two difficulties with this direct approach when $|\, x\, |$ is large. First, cancellation can occur (just as it did in the calculation of $e^{-10}$ in Section 1.3) because intermediate terms in the sum have magnitudes much larger than that of the final result. Second, in order to make the truncation error, $E_n(x)$, small, we must use large values of $n$. Both difficulties can be overcome through a process called *range reduction*. In particular, we can always choose an integer $k$ such that $y = x - \pi k$ lies in the range $[\,0, \pi\,]$. The values of sin($x$) and sin($y$) are then simply related by the trigonometric identity

$$\sin(x) = (-1)^k \sin(\, x - \pi k).$$

In Figure 6 we have marked $x = 13\pi/8$. Using the rule above, it can be reduced to $y = 5\pi/8$ (which is also marked) because $\sin(13\pi/8) = -\sin(5\pi/8)$.

In addition, we note that sin($x$) is symmetric about $\pi/2$, so

$$\sin(x) = \sin(\,\pi - x).$$

For example, in the above figure we see that $5\pi/8$ can be reduced further to $3\pi/8$.

In summary, for any input $x$ we need only compute the appropriate value of $k$ so that $y$ is between 0 and $\pi$ and $y = x - \pi k$. If this value of y is larger than $\pi/2$ then it can be replaced by its ‘‘reflected’’ value $\pi - y$ (i.e. reflected about $\pi/2$). Then sin($x$) is given approximately by

$$\sin(x) \approx (-1)^k S_n(y),$$

where $y$ now lies in the range $[\,0, \pi/2\,]$. By choosing $n$ sufficiently large we can make the error, $|\, E_n(y)\, |$, as small as we like for all $y$'s in the range $[\,0, \pi/2\,]$. This algorithm is used in the following program.

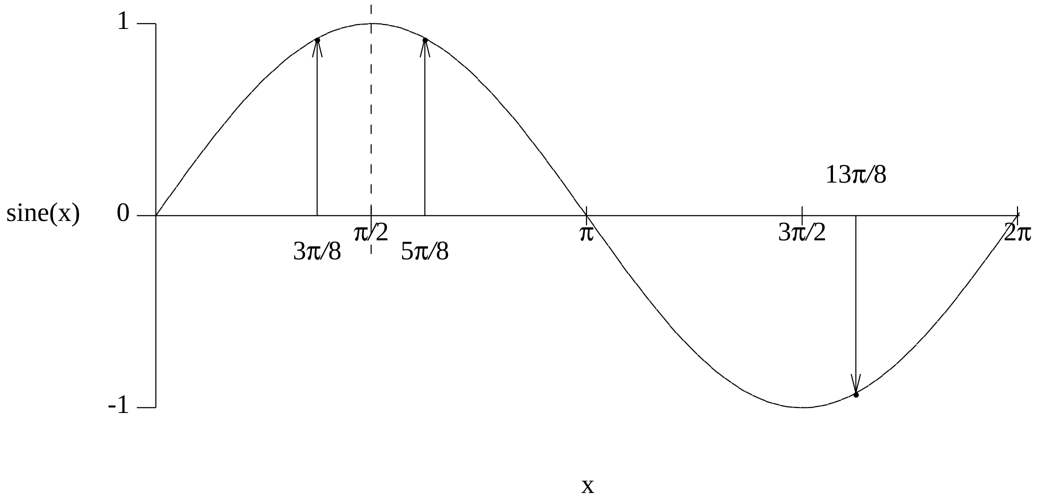



Figure 6. Range reduction to $[\,0, \pi/2\,]$.

```
1   function sine (x : real) : real
2      % Polynomial approximation to sin(x), using a range reduction
3      % to [0, pi/2].  If abs(x) <= pi then the absolute error is less
4      % than 5e-8, otherwise the error grows with abs(x).
5
6      % Range reduction to [ 0, pi ].
7      const pi := 3.1415927
8      var sgn := sign (x)
9      var y := abs (x) mod pi
10     if ((abs (x) div pi) mod 2) > 0 then
11         sgn := – sgn
12     end if
13     % Further range reduction to [ 0, pi/2].
14     y := min (y, pi – y)
15     % Compute polynomial approximation to sin(y).
16     if y < 1e-4 then
17         % Use sine series truncated after first term.
18         result sgn * y
19     else
20         % Use sine series truncated after 7th term.
21         const z := y * y
22         const poly : real := ((((((1.6059044e–10 * z – 2.5052108e–8) * z – 2.7557319e–6) * z
23                               – 1.9841270e–4) * z – 8.3333333e–3) * z – 1.6666667e–1) * z + 1) * y
24         result sgn * poly
25     end if
26  end sine
```

It is convenient to work with $|x|$, and, since $\sin(-x) = -\sin(x)$, the program can easily look after the case when $x < 0$. The program assigns sign($x$) to the variable *sgn* (line 8), and then uses this value in the calculation of the approximation to sin($x$) (see line 18 or 24). The range reduction to $[0, \pi]$ is performed on $|x|$ (lines 9 through 12), and the sign of *sgn* is reversed if $|x|$ and $y$ differ by an odd multiple of $\pi$.

If $y$ is extremely small, an underflow could occur when $z$ is evaluated, or, if not at that point, then a little later when $z$ is multiplied by the first constant in the polynomial used to evaluate the final result. However, when $y$ is so small that any such underflow would occur, sin($y$) can be accurately approximated by the first term in the sine series, namely $y$. The program uses this approximation if $y < 10^{-4}$ (line 18), but otherwise uses the polynomial $S_7(y)$ (lines 20-24).

Assuming eight-digit arithmetic as before, a careful analysis of all roundoff and other errors affecting the final result will show that, if the original value of $x$ is in the interval $(-\pi, \pi)$, the result will be in error by at most $5e-8$. When $x$ is outside the interval $(-\pi, \pi)$, an approximation to $\pi$, namely the constant *pi*, must be used, and this can introduce additional error in the final result. It would be better if the approximation to $\pi$ and the range reduction calculations could be done in higher precision. (In Sun systems, the default value of $\pi$ is accurate to 66 bits, but provision is made for an option of much higher accuracy.) Otherwise the additional error must be kept in mind. It can be quite large if $|x|$ is large.

Finally we note that the range could have been reduced further to $[0, \pi/4]$ through the use of the trigonometric identity

$$\sin(y) = \cos(\pi/2 - y).$$

To use this, the above program should be modified so that whenever the reduced value $y$ lies in the interval $(\pi/4, \pi/2)$, then $y$ is replaced by $\pi/2 - y$, and the cosine Taylor series is used to approximate sin($y$). When this is done, it can be shown that we need only five terms from the Taylor series of sine and cosine to get roughly the same accuracy that the above program achieves using seven terms.

### 1.5.3. Exercises

1. To show how rapidly the square root iteration converges, substitute $x_n = \sqrt{a}\,(1 + \varepsilon_n)$ into Heron's formula and prove that:

$$\varepsilon_{n+1} = \frac{(\varepsilon_n)^2}{2\,(1 + \varepsilon_n)}$$

(This means, for example, that, if $a_0$ has a relative error of no more than .1, then $a_1$'s error will be less than .005, $a_2$'s less than .00002, and $a_3$'s less than .0000000002, except that, in our eight-digit arithmetic, rounding errors dominate the total error in the final iteration. The rounding errors would have to be carefully analyzed in order to guarantee the accuracy of the final approximation.)

## 1.6. Numerical Integration

Another problem that often arises in numerical computation is that of finding an approximation to a definite integral, say

$$I = \int_a^b f(x)\,dx.$$

A method for doing this is called *numerical integration*, or *numerical quadrature*.

Such methods are almost always based on dividing the interval of integration $[a,\ b\,]$ into a number of subintervals $[x_{i-1},\ x_i],\ \ i = 1,2,\ \cdots,n,$ where $a = x_0 < x_1 < x_2 \cdots < x_n = b,$ then calculating an approximation to the integral over each subinterval, and finally adding together all these individual approximations to obtain a total approximation to the entire integral. One formula that could be used for the approximation over $[x_{i-1},\ x_i]$ is the mid-point formula

$$f((x_{i-1} + x_i)/2)\,(x_i - x_{i-1});$$

another is the trapezoidal formula

$$(f(x_{i-1}) + f(x_i))\,(x_i - x_{i-1})/2.$$

A decision would have to be made, by the user or by the program, as to what subinterval sizes to use. For example, the points of subdivision could be equally spaced, so that $x_i - x_{i-1} = h,\ \ i = 1,2,\ \cdots,\ n,$ where $nh = b - a,$ for some value of $h$. Then only $h$ needs to be chosen.

To choose $h$, the user, or the program, could try a sequence of values, each say half of its predecessor, until the values of the approximations are not changing significantly, and then accept the final value as the required approximation to the integral.

For a general-purpose library program, it may be better not to use equally spaced points. If the integrand is very smooth in some places, but quite wiggly in others, it is probably better to use smaller subintervals near the wiggly parts.

A program that chooses the subinterval sizes so that they are *adapted* to the local behaviour of the integrand, as suggested here, is called an *adaptive* program. A specific way in which such an adaptive program can be implemented will now be described.

The general idea is to have the program begin with a relatively small number of equal subintervals, and then calculate an approximation to the integral over each of them, along with an estimate of the error in each of these individual approximations. The sum of the individual error estimates is the estimate of the total error, i.e., of the error in the approximation to the total integral, which itself is obtained as the sum of the individual approximations.

If this estimate of the total error is not small enough (i.e., if its magnitude is greater than a tolerance specified by the user), then the program locates the subinterval with the largest individual error estimate, and divides it into two equal subintervals. Individual approximations and error estimates for these two new subintervals are obtained, the corresponding information from the old subinterval is discarded, and the estimate of total error is updated accordingly.

If the new estimate of total error is still not small enough, the process of locating the subinterval with the largest individual error estimate, subdividing it, and so on, is carried out again. This is done repeatedly until the total error estimate is satisfactorily small, or until some upper bound on the number of subdivisions permitted is reached.

To implement such an adaptive program, we need to have formulas for calculating approximations and error estimates over individual subintervals. One relatively simple formula for the approximation over $[x_{i-1},\ x_i]$ is

$$a_i = (f(x_{i-1}) + 2f((x_{i-1} + x_i)/2) + f(x_i))(x_i - x_{i-1})/4,$$

which results from applying the trapezoidal rule twice, once to each half interval. A corresponding rule for the error estimate is

$$e_i = (-f(x_{i-1}) + 2f((x_{i-1} + x_i)/2) - f(x_i))\,(x_i - x_{i-1})/12.$$

This is not as easy to derive, but it can be shown mathematically that the ratio of $e_i$ to the corresponding true error approaches 1 as $x_i - x_{i-1}$ approaches 0. (Think of $x_{i-1}$ being fixed as $x_i - x_{i-1} \to 0$. For the proof of this result it is sufficient to assume that $f''(x)$ exists and is continuous in a neighborhood of $x_{i-1}$.)

The program can keep track of the information associated with each subinterval in a record data structure. One possibility is to store the following in the record associated with $[x_{i-1}, x_i]$:

$$x_{i-1}, \quad x_i - x_{i-1}, \quad f(x_{i-1}), \quad f((x_{i-1} + x_i)/2), \quad f(x_i), \quad a_i, \quad e_i$$

If the records are organized into a heap, using $|e_i|$ as the key so that the record whose $|e_i|$ is largest is at the top of the heap, then the time needed to search for the next subinterval to be further divided is essentially eliminated. Once this subinterval is divided, and its record is replaced by two new ones, the heap property can be restored quite quickly – two processes are required, and each requires at most $\log_2 n$ comparisons.

To get started, the program must be given an initial number of subintervals. The user could provide this, but it is better to do so indirectly, by providing the initial number of function evaluations to be used, and letting the program determine the corresponding initial number of subintervals. In this way, the user is asked to provide information only about how frequently the integrand should be sampled initially. The user does not need to know that the program uses about two values of the integrand per subinterval; in fact, another program might be based on formulas for each subinterval which use many more values per subinterval. The user should not have to know about such things.

It is also reasonable to expect the user to put some sort of upper bound on how long the subdividing process should be allowed to continue before the program gives up. One way to do this is to require the user to provide an upper bound on the number of integrand evaluations the program is allowed to make. This maximum number is a reasonable measure of how much work the program is allowed to do before giving up, and, as with the initial number of evaluations, does not require the user to be particularly aware of what formulas are being used by the method. The parameter list for the program might therefore include each of the following

| | | |
|---|---|---|
| *a, b, f, tolerance* | – | needed to specify the problem |
| *initevals, maxevals* | – | as explained above |
| *approx, errest* | – | returned by the program |

Here *f* is a function name (the user must provide the function for evaluating *f*), *approx* is the approximation to the integral which is calculated by the program, and *errest* is the estimate of the error in this approximation, which is also calculated by the program. Note that the condition $|errest| > tolerance$ indicates that the program had to return, before achieving $|errest| \le tolerance$, because *maxevals* was about to be exceeded.

**Exercises**

**1.** If the trapezoidal rule is applied directly to

$$I = \int_a^b f(x)dx,$$

with $n$ equal subdivisions of width $h$ (where $nh = b - a$) to give an approximation $A$, it can be shown that the truncation error is

$$I - A = -h^2(b-a)f''(\xi)/12$$

where $\xi$ is some number in [a, b], provided it is assumed that $f''$ exists and is continuous on [a,b].

Suppose that such a method is to be used to evaluate

$$\int_1^{10} \frac{dx}{x}$$

with an error of no more than .001. Find a value of $h$ that is small enough (but not too small!) to ensure that the truncation error will contribute no more than half of this amount. How much precision would be needed to ensure that roundoff errors also did not contribute more than half?

**2.** Write a program to implement the trapezoidal rule with fixed subinterval width, and try it with several different widths. As the widths get smaller, the approximations should approach a limit. Some examples to be tried are

$$\int_1^{10} \frac{dx}{x}, \quad \int_0^1 \frac{dx}{1+x^2}, \quad \int_0^1 e^{-x^2} dx, \quad \int_1^2 \ln x \, dx.$$

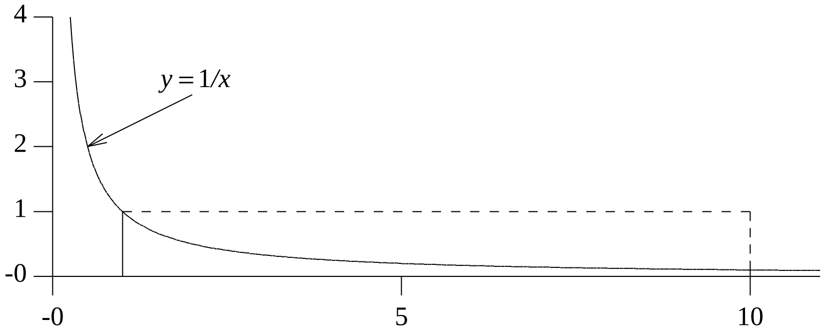
How much confidence do you have in the final results? (Would you fly in an aeroplane whose reliability depended on the accuracy of your results?)

**3.** Write an adaptive quadrature subprogram based on the ideas described in this section, and provide good documentation for a potential user. (Small points: the subprogram should work properly if $a = b$, or if $a > b$, or if the integrand is negative. It should do something sensible if *maxevals* $\leq$ *initevals*.)

**4.** A very different approach to approximating integrals involves what are rather colourfully called Monte Carlo methods. (Recall the related discussion of Monte Carlo methods of simulation in Chapter 2.) As an example, consider again the integral

$$\int_1^{10} \frac{dx}{x}.$$

To approximate the value of this integral, we generate a large number, say $n$, of random points in the rectangle $1 \leq x \leq 10$, $0 \leq y \leq 1$:




We count the number of these points which fall below the graph of $y=1/x$. This number is a fraction of the total number, and this is taken to be the fraction of the area of the rectangle that lies below $y=1/x$, which then yields the desired approximation to the integral.

Write a program to implement this method and obtain approximations with, for example, $n = 100,\ 1000,\ 10000$.

The main difficulty with this method is that it converges so slowly. (It can be shown that the error decreases in proportion to $1/\sqrt{n}$, so each extra decimal place of accuracy requires 100 times as many random points.) However, there are situations in which it is quite useful, especially with multi-dimensional integrals.

## 1.7. Non-Linear Equations

There are many methods, and many variations on individual methods, for solving non-linear equations. But there is one class of methods, known as Newton methods, that is most important. One reason for their importance is that they are very general − they can be applied to *systems* of non-linear equations − but we will consider applying them here only to single, scalar equations.

As simple examples, we can consider finding solutions to equations such as

$$e^{-x} = x, \quad cos(x) = x^3.$$

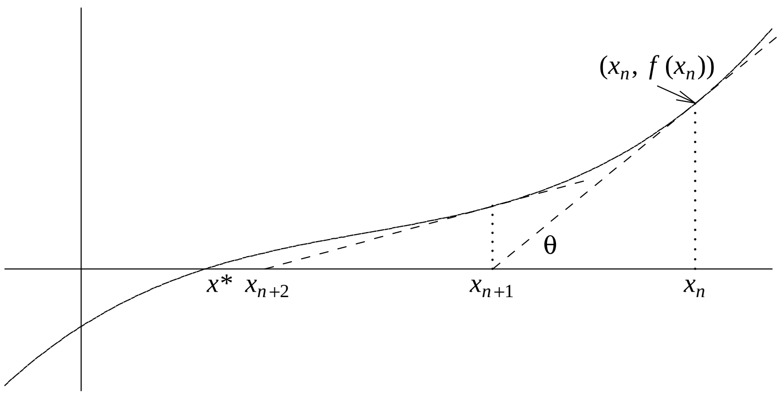
Any such equation can be written as

$$f(x) = 0.$$

For example, in the first case $f(x)$ is given by:

$$f(x) = e^{-x} - x.$$

An easy way to develop Newton's basic formula, is to start with a graph of $y=f(x)$. Then suppose $x_n$ is an approximation to the solution, $x^*$. The next approximation, $x_{n+1}$, is obtained, as shown in the diagram, by drawing a tangent to the graph at $(x_n,\ f(x_n))$ and noting where is crosses the x-axis. You can imagine this step being repeated, first with $x_{n+1}$ in place of $x_n$, then with $x_{n+2}$ in place of $x_{n+1}$, and so on. From the graph it would appear that this sequence $x_n,\ x_{n+1},\ x_{n+2},\ \ldots,$ approaches $x^*$ quite rapidly.




A formula for $x_{n+1}$ in terms of $x_n$ is easily found from the graph. If θ is the angle made by the tangent with the x-axis, we have

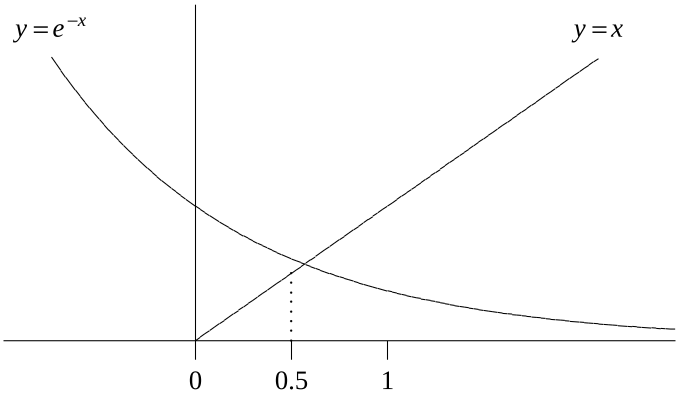
$$f'(x_n) = tan(\theta) = f(x_n)/(x_n - x_{n+1}).$$

Solving for $x_{n+1}$, we obtain

$$x_{n+1} = x_n - f(x_n)/f'(x_n).$$

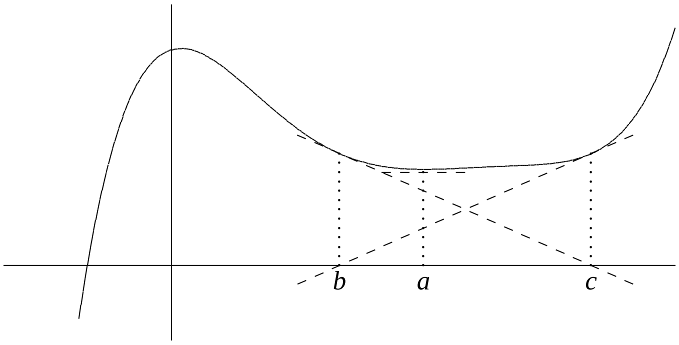
A method based on this formula must have some way of getting started (i.e., selecting $x_0$). Provision must also be made for stopping ( $x_{n+1}$ close to $x_n$, or $f(x_n)$ small, or a combination of the two, etc.), and also for what to do if $f'(x_n)$ becomes too small. There is no good general way of making all these decisions. However, in specific cases it is usually quite easy.

For example, consider the solution to $e^{-x} = x$. A very rough graph comparing the functions $y = e^{-x}$ and $y = x$ shows that they cross near $x = 0.5$, and this value can be used for $x_0$. However, in general the choice of $x_0$ can be quite difficult, especially when systems of equations are involved. What is done usually depends very much on the application.




Stopping is also easy in specific situations. Convergence is usually very rapid, once an approximation close to the solution is obtained. In fact, there are rigorous mathematical theorems to the effect that such convergence will eventually become *quadratic* if $f'$ is bounded away from zero in a neighbourhood of the solution. This means that, once 1 digit of accuracy is obtained, successive applications of the formula will produce first 2, then 4, then 8, and so on, digits of accuracy.

Convergence at first may not be nearly so good. In fact, the method may not converge at all. The figure below shows two simple examples to illustrate ways in which convergence may fail. First, if $x_n = a$, then the tangent line is horizontal and no new estimate can be found (we are at a local minimum of the function $f$). Second, if $x_n = b$ (or $c$), then we are caught in an infinite loop where the function value is not changing.




A general purpose program might try to monitor the iterations, and then restart in a somewhat arbitrary way if $f'(x_n)$ becomes too small compared to $f(x_n)$, or if successive values of $f$ do not seem to be decreasing.

We conclude this section with brief mention of a special problem, namely the problem of finding an approximation to $\sqrt{a}$, for a given $a > 0$. This can be expressed in terms of solving the non-linear equation

$$f(x) = x^2 - a = 0.$$

Applying Newton's formula to this equation yields

$$\begin{aligned} x_{n+1} &= x_n - (x_n^2 - a)/2x_n \\ &= (2x_n^2 - x_n^2 + a)/2x_n \\ &= (x_n + a/x_n)/2 \end{aligned}$$

This is *Heron's formula*, which was used in Subsection 1.5.1 as the basis for constructing a square root subprogram. It happens that the sequence generated by this formula will converge to $\sqrt{a}$ for any $x_0 > 0$. (Of course, in practice roundoff errors will eventually interfere, but accuracy to within a few units in the last place is easily achieved, even in the presence of roundoff errors.) An important consideration is to get off to a good start, i.e., with a good value of $x_0$. it was shown in Subsection 1.5.1 how a value of $x_0$ could be found quite quickly which already has a relative error of less than $10^{-1}$.

**Exercises**

**1.** Find a solution of

$$e^{-x} = x$$

with an error of less than $10^{-6}$.

**2.** Use Heron's formula to find an approximation to $\sqrt{2}$ with an error of less than $10^{-6}$. Do the arithmetic on a calculator, so that you will be able to observe how rapidly the method converges. Try $x_0 = 1$. What happens if this starting approximation is not nearly so good, say $x_0 = 100$?

**3.** Find two or three of the smaller roots of

$$x = tan\,(x)$$

What are the roots when $x$ is large?

**4.** Find a formula, analogous to Heron's formula, for calculating approximations to cube roots.

**5.** Develop a general subprogram for calculating approximations to solutions of

$$f(x) = 0$$

Input parameters should be $x_0$ and the two functions $f$ and $f'$. The program should try to do something sensible if a value of $f'$ is too small. At the very least it should return with a parameter set to indicate that this difficulty has arisen, rather than simply allowing the program to fail, for example, by an attempt to divide by zero.

## 1.8. Systems of Linear Equations

Problems involving matrix-vector calculations of various kinds turn up in many application areas, very often as part of a larger problem, such as solving differential equations, certain statistical calculations, or applying a Newton method to systems of non-linear equations. It is therefore very important to have good library programs for the matrix-vector calculations that arise in practice.

We will concentrate on the problem of solving systems of linear equations. We write the system as follows:

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1, \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2, \\
&\;\;\vdots \\
a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n.
\end{aligned}$$

One approach to solving this system is to use systematic *elimination* and *back-substitution*. If we suppose for the moment that $a_{11} \neq 0$, we can see that subtracting a multiple $(a_{21}/a_{11})$ of the first row from the second will cause the $x_1$ term to be eliminated from the second equation. Similarly, the $x_1$ term can be eliminated from the third equation, then the fourth, and so on, until the $x_1$ terms have been eliminated from all equations after the first. In general,

$$a_{jk}^{(1)} = a_{jk} - \frac{a_{j1}\, a_{1k}}{a_{11}} \quad \text{for } j=2,...,n \text{ and } k=1,...,n.$$

and in particular,

$$a_{j1}^{(1)} = 0 \quad \text{for } j=2,...,n.$$

If the coefficient of $x_2$ in the remaining second equation is not zero, a similar process can be used to eliminate the $x_2$ terms from all equations after the second, then the $x_3$ terms from all equations after the third, and so on. When $a_{ii}^{(i-1)}$ is used to set coefficients $a_{ji}^{(i)}$ to 0, the other equations are modified by:

$$a_{jk}^{(i)} = a_{jk}^{(i-1)} - \frac{a_{ji}^{(i-1)}\, a_{ik}^{(i-1)}}{a_{ii}^{(i-1)}} \quad \text{for } j=i+1,...,n \text{ and } k=j,...,n.$$

Finally there remains only a triangular system of the form:

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + a_{13}x_3 +...+ a_{1n}x_n &= b_1, \\
a_{22}^{(1)}x_2 + a_{23}^{(1)}x_3 +...+ a_{2n}^{(1)}x_n &= b_2^{(1)}, \\
a_{33}^{(2)}x_3 +...+ a_{3n}^{(2)}x_n &= b_3^{(2)}, \\
&\;\;\vdots \\
a_{nn}^{(n-1)}x_n &= b_n^{(n-1)}.
\end{aligned}$$

If $a_{nn}^{(n-1)} \neq 0$, the last of these equations can be solved for $x_n$. The result can then be substituted back into the second last equation, which can then be solved for $x_{n-1}$, and so on, back through all the equations, for $x_{n-2}$, $x_{n-3}$, . . . , $x_1$.

The values we have had to assume are not zero ($a_{11}$ when eliminating $x_1$ terms, $a_{22}^{(1)}$ when eliminating $x_2$ terms, ... , $a_{n-1,n-1}^{(n-2)}$ when eliminating the $x_{n-1}$ term, and finally $a_{nn}^{(n-1)}$ when solving for $x_n$) are called the *pivots*. If a particular diagonal element, other than the last one, that we want to use as a pivot happens to be zero, we can search the column below it for a non-zero element, and then interchange the two corresponding equations in order to get a non-zero element in the pivot position. (Interchanging equations in a system of equations does not change the solution.)

Mathematically, this interchanging of equations to ensure non-zero pivots is all that has to be done. If a situation is ever reached where the diagonal and all elements below it are all zero, or where the last pivot is zero, we know that the system is *singular* – which means it either has no solution at all or an infinite number of solutions – and the process can be stopped with an indication that the problem is singular.

However, in practical terms, it is not enough to simply ensure that the pivots are non-zero. They should be relatively large in some sense, in order to ensure that the multipliers are relatively small, since this helps to reduce the effect of roundoff errors. (We will not try to analyse this in more detail, but just accept the fact.) What is usually done is to search the column of coefficients, from the diagonal down to the bottom, for the largest element in absolute value, and then use that one as the pivot. This is called *partial pivoting*. (*Complete pivoting* includes other columns to the right as well, and requires interchanging columns as well as rows. It is generally considered that the extra work is not worthwhile.)

Because of roundoff errors, the distinction between singularity and non-singularity cannot be made as sharp in practice as it can when the arithmetic is assumed to be exact. Small pivots may be an indication of trouble in practical applications.

A useful measure of the efficiency of the elimination and back-substitution process is the total number of multiply operations required. To eliminate the $x_i$ term in the *j-th* equation requires $n-i+1$ multiplies (the multiplier is $a_{ji}^{(i-1)}/a_{ii}^{(i-1)}$, and it is used to multiply each of $a_{i,\,i+1}^{(i-1)}$, $a_{i,\,i+2}^{(i-1)}$, $\cdots$ $a_{in}^{(i-1)}$, $b_i^{(i-1)}$). This has to be done for $j = i+1$, $i+2$, $\cdots$ , $n$, i.e., $n-i$ times. The total number of multiplies to eliminate the $x_i$ terms is therefore $(n-i+1)\,(n-i)$. This must be done for $i = 1, 2, \cdots, n-1$. The total number of multiplies for the elimination part of the process is therefore $\sum_{i=1}^{n-1} (n-i+1)(n-i) = n^3/3 + O(n^2)$. It is fairly easy to show that the back-substitution requires only another $n^2/2 + O(n)$ multiplies, so the grand total of all multiplies is $n^3/3 + O(n^2)$.

It is interesting to compare this with the cost of simply multiplying two $n \times n$ matrices, which obviously requires $n^3$ multiplies (assuming the matrix multiplication is done in the usual straightforward manner).

Solving linear systems is an example of a problem which, in itself, involves no truncation error. (It may, however, be an approximation to some other problem − such as a differential equation − so that a truncation error has already been incurred.) An error analysis of the elimination and back-substitution process therefore involves only an analysis of the cumulative effects of roundoff errors. We will not attempt a complete analysis, but we will try to give some indication of the basic kind of result to be expected, since the nature of this result − the result of a so-called *backward* error analysis − is of considerable importance.

We concentrate on analysing in a very general way the cumulative effect of errors in a sum. We start with the fundamental rule, which states that, in the absence of overflow or underflow,

$$fl(a+b) = (a+b)(1+\varepsilon), \quad |\varepsilon| \le u,$$

where $u$ is a small quantity known as the relative roundoff error bound. (For example, with the 6-digit arithmetic used earlier, $u = .5 \times 10^{-5}$.) In words, this rule states that the result, $fl(a+b)$, of adding the floating point numbers, $a$ and $b$, in floating point arithmetic, is the mathematically correct sum, $a+b$, except for a factor $1+\varepsilon$, where $\varepsilon$ is some small quantity whose magnitude is less than or equal to $u$.

The rule holds for all arithmetic operations, not just addition. The value of $u$ is a characteristic of the arithmetic; it is *double that amount if the arithmetic is chopped.*

A convenient notation is to let $\varepsilon$ represent any small quantity such that $|\varepsilon| \le u$, so that different occurrences of $\varepsilon$ need not have the same value. Thus, instead of noting that $fl(abc) = (abc)(1+\varepsilon_1)(1+\varepsilon_2)$, we can write $fl(abc) = (abc)(1+\varepsilon)^2$, and keep in mind that the two occurrences of $\varepsilon$ implied in $(1+\varepsilon)^2$ need not be the same.

Now, it is tempting to write $fl(a+b+c) = (a+b+c)(1+\varepsilon)^2$, but this is totally wrong, since there can be cancellation in the computation of the sum (try $a = 1$, $b = 10^{-8}$, $c = -1$ in 6-digit arithmetic), and this illustrates the key idea that leads to the so-called *backward* error analysis.

Being more careful, we can write

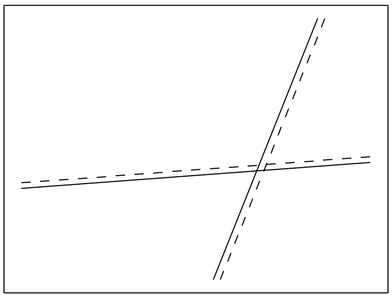
$$\begin{aligned} fl((a+b)+c) &= (fl\,(a+b)+c)(1+\varepsilon) \\ &= ((a+b)(1+\varepsilon)+c)(1+\varepsilon) \\ &= a(1+\varepsilon)^2 + b(1+\varepsilon)^2 + c(1+\varepsilon) \\ &= \hat{a} + \hat{b} + \hat{c} \end{aligned}$$

where $\hat{a} = a(1+\varepsilon)^2$, $\hat{b} = b(1+\varepsilon)^2$ and $\hat{c} = c(1+\varepsilon)$. Thus we can say that the summing (in floating point arithmetic) of $a$, $b$ and $c$, in that order, produces a result which is the *exact* sum of three quantities, $\hat{a}$, $\hat{b}$ and $\hat{c}$ which differ, respectively, by only very small quantities from the original $a$, $b$ and $c$. The sum obtained may differ considerably from the true sum, but it will nevertheless be the exact sum for a problem which differs from the original by only very small quantities.

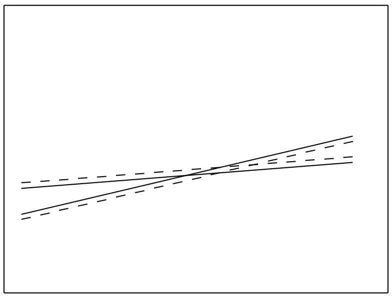
The result can obviously be generalized to sums of more than three numbers. The result can also be generalized to solving systems of equations (although this generalization is certainly not obvious). In fact, it can be shown that the solution of a linear system (in floating point arithmetic) is the exact solution of another linear system that differs from the original only by very small quantities.

This is an example of a *backward* error analysis. The errors are, in effect, thrown backward onto the original problem, and are shown to be equivalent to only small changes in the original problem.

So, the solution obtained satisfies the original system of equations almost exactly. This may be all that is required. On the other hand, we may want to know how close the solution obtained is to the exact solution of the original problem. But this depends on what is called the *condition* of the original problem. This can be illustrated with two examples:



Well-Conditioned



Badly-Conditioned

The first example shows graphs of two equations in two unknowns − and it is clear that small changes in these equations will cause only small changes in the solution. This problem is *well-conditioned*, and the computed solution will be close to the true solution. On the other hand, it is obvious that small changes in the two equations of the second example could result in large changes in the solution. This example is *badly-conditioned*, and the computed solution may differ considerably from the true solution.

A backward error analysis cannot be so successfully applied to all problems in linear or nonlinear algebra (for example, a corresponding result cannot be found for matrix inversion). However, the technique can be applied successfully to many important methods, and is extremely helpful as a basis for understanding and assessing the accuracy and usefulness of those methods.

**Exercises**

**1.** Develop a subprogram for carrying out the elimination and back-substitution process described in this section. Careful testing and good user documentation are also required.

**2.** The value of the determinant

$$\begin{vmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ . & & & \\ . & & & \\ a_{n1} & a_{n2} & \ldots & a_{nn} \end{vmatrix}$$

is not changed by subtracting a multiple of one row from another. Interchanging two distinct rows of a determinant merely changes its sign. The value of the triangular determinant

$$\begin{vmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ 0 & a_{12} & \ldots & a_{2n} \\ . & & & \\ . & & & \\ 0 & 0 & \ldots & a_{nn} \end{vmatrix}$$

is the product of its diagonal elements. These three rules are enough to define, at least mathematically, the value of a determinant. In practice, partial pivoting should be used in the process of triangularizing the determinant, in order to reduce the effect of roundoff errors. Develop, test and document a subprogram for evaluating determinants.

**3.** An alternative definition for the value of a determinant is as follows. A determinant of order 1, a single element between vertical bars, say $|a_{11}|$, is defined to be the value of that element, $a_{11}$ in this case. Then determinants of order $i$ can be defined recursively in terms of determinants of order $i-1$ by means of the following rule:

$$\begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1i} \\ a_{21} & a_{22} & \cdots & a_{2i} \\ \vdots & & & \\ a_{i1} & a_{i2} & \cdots & a_{ii} \end{vmatrix} = \begin{array}{rcl} & a_{11} \times & \text{(determinant of order } i-1 \text{ obtained from determinant of order } i \text{ by removing row 1 and column 1)} \\ - & a_{12} \times & \text{(determinant of order } i-1 \text{ obtained from determinant of order } i \text{ by removing row 1 and column 2)} \\ + & a_{13} \times & \text{(determinant of order } i-1 \text{ obtained from determinant of order } i \text{ by removing row 1 and column 3)} \\ \vdots & & \\ + (-1)^{i-1} & a_{1i} \times & \text{(determinant of order } i-1 \text{ obtained from determinant of order } i \text{ by removing row 1 and column } i\text{).} \\ \vdots & & \end{array}$$

How many multiply operations are required to evaluate a determinant of order $n$ according to this definition? How does this compare with the method in the previous question? At 1 million multiplies per second, how long would it take to evaluate a determinant of order 20 by each of these two methods?