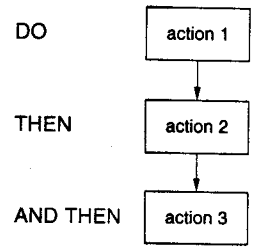


Fig. 2.2 A simple (linear) sequence of actions.

2.8 METHOD: STRUCTURED PROGRAMMING

You should always be striving to build *structures* into your program that clearly reveal the content and logic of the program. The physical structuring of your code can be built by using successive indentations for different sections (ignored by the compiler); by the frequent and judicious use of comments and spacings; and by using upper- and lowercase letters to improve clarity (the Fortran compiler is case-insensitive). You may, nonetheless, wish to avoid non-commented blank lines because they will not always be accepted by compilers.

On a more conceptual level, modern high-level languages contain building blocks to provide structure to programs; they have been proven to be logically sufficient for all programming needs. The elements that are used to construct the logical building blocks are given in Fig.2.3. Some common structures are illustrated in Figs.2.2–2.7. These logical building blocks start with the linear sequence, shown in Fig. 2.2, and include:

Repeat N times, Fig.2.4: *Do* all instructions up to the end of loop indicator *Endo* N times.

If-then-else, Fig.2.5: *If* a certain condition is met, *Then* execute some instructions, or *Else*, do something else. When one of these possibilities is finished, this sequence ends.

Repeat unknown number of times, Fig.2.6: *While* some condition is met, keep repeating these instructions. *If* the condition is no longer met, *go to* statements beyond *Endwhile*.

You will note that these structures have well-defined beginnings, endings, and conditions for their actions, all of which help clarify the program flow. Of

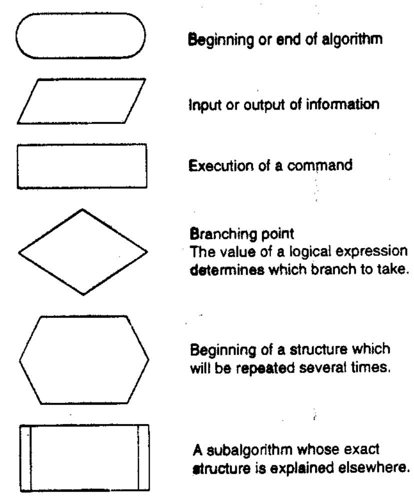


Fig. 2.3 The logical elements used to construct the requisite logical structures of programs.

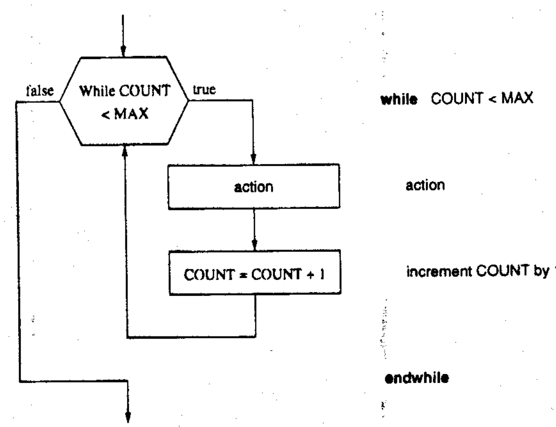
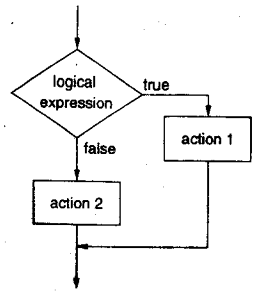
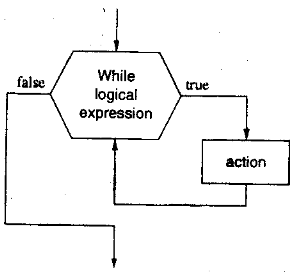


Fig. 2.4 The *Repeat Loop MAX times* structure and pseudocode. This is a special case of the *While* loop.




```
if logical expression is true
   then action1
   else action 2
endif
```

Fig. 2.5 The *If-then-else* structure and pseudocode.




```
while logical expression is true
   action
endwhile
```

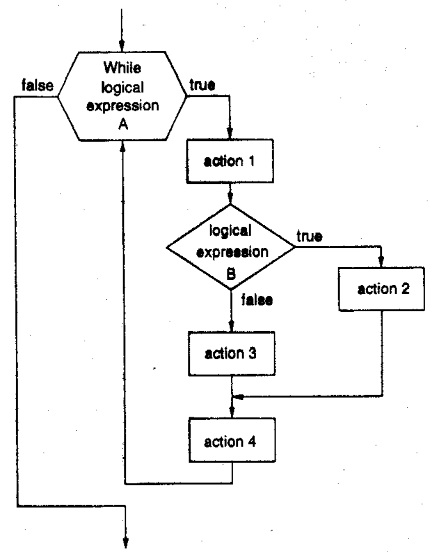
Fig. 2.6 The *While* structure and pseudocode.

course, actual programs often contain more complicated logic, yet as we see in Fig.2.7, these basic structures may be combined to provide a richer structure.

2.9 METHOD: PROGRAMMING HINTS

Some specific programming hints that may help you implement the preceding general rules for writing programs are

- Always keep an updated, working version of your program; make modifications on a copy.
- Use the standard version of the programming language if you want to *port* your code to another computer or immediately run it when future systems become available. (Avoid local language extensions.)
- Add plenty of comments and documentation as you write the code, with at least a short description about each subprogram. This will help keep




```
while logical expression A is true
   action 1
   if logical expression B is true
      then action 2
      else action 3
   endif
   action 4
endwhile
```

Fig. 2.7 A combination of *While* and *If-then-else* structures.

your mind on track and be useful later (a natural action for those who talk to themselves or keep a diary).

- Use descriptive names for variables and subroutines, like *mass* and *temp*, and keep them similar to the variables employed in standard texts and papers. Describe your variables in comment statements.
- Declare all variables (do not use the *Implicit* statement in Fortran unless you make it an *Implicit none*). Combined with cross-referenced maps, this helps you pick up spelling, typographical, and forgetfulness errors.
- Avoid global variables.
- In Fortran, use statement labels only for *Continue* and *Format* statements.
- Avoid *Equivalence* statements. They make the logic too hard to follow.
- Remember that compilers make errors, too. You'll want to be particularly careful if your programming is subtle or clever or highly convoluted. Comparing results derived with different levels of *optimization*, different flags, or reorganized program parts may help reveal compiler bugs (the

least-optimized answer is usually your best bet). Using program checkers like *lint* is also recommended. And if you are fortunate enough to have different machines around, you can try different compilers.

2.10 PROBLEM 2: LIMITED RANGE OF NUMBERS

Computers may be powerful, but they are finite. A **problem** in computer design is how to represent a general number in a finite amount of space, and then how to deal with the approximate representation that results.

2.11 THEORY: NUMBER REPRESENTATION

If we are given the digits 0 and 1 as the microscopic units of memory (*bits*), it should be no great surprise that all numbers are ultimately represented in *binary* form. Correspondingly, there are only 2^N integers that can be represented with N bits. Because the sign of the integer is represented by the first bit (a zero bit for positive numbers), this leaves the remaining $N-1$ bits to represent the value of the integer. Therefore N-bit integers will be in the range $[0, 2^{N-1}]$. Already we begin to see the limitations.

Long strings of zeros and ones are fine for computers, but are awkward for people. Consequently, binary strings are converted to *octal*, *decimal*, or *hexadecimal* numbers before results are communicated to people. Octal and hexadecimal numbers are nice because the conversion loses no precision, but not so nice because our decimal rules of arithmetic do not work for them. Converting to decimal numbers make the numbers easier for us to work with, but we often lose precision.

A description of a particular computer system will normally state the number of bits used to store a number (also called *word length*). This word length is often expressed in *bytes*, where

$$1\text{byte} \equiv 1\text{B} \stackrel{\text{def}}{=} 8\text{bits}. \tag{2.1}$$

Conventionally, storage size is measured in bytes or kilobytes. Be careful, not everyone means the same thing by a thousand:

$$1\text{K} \stackrel{\text{def}}{=} 1\text{KB} = 2^{10}\text{bytes} = 1024\text{bytes}. \tag{2.2}$$

This is often (and confusingly) compensated for when memory size is stated, for example

$$512\text{K} = 2^9\text{bytes} = 524,288\text{bytes} \times \frac{1\text{K}}{1024\text{bytes}}. \tag{2.3}$$

Conveniently, 1 byte is also the amount of memory needed to store a single character, like the letter "a" or "b." This adds up to a typical typed page requiring ~3 KB.

The memory chips in some of the older personal computers used 8-bit words. This means the maximum integer was $2^7 = 128$ (7 because one bit is used for the sign). Trying to store a number larger than possible (*overflow*) was common on these machines, sometimes accompanied by an informative error message and sometimes not. At present, most workstation-class computers use 32 bits for an integer, which means that the maximum integer is $2^{31} \simeq 2 \times 10^9$. While at first this may seem a large range for numbers, it really isn't compared to the range of sizes encountered in the physical world. For example, the ratio of the size of the universe to the size of a proton is 10^{24}.

2.12 METHOD: FIXED AND FLOATING

Real numbers are represented on computers in either *fixed-point* or *floating-point* notation. In fixed-point notation, the number x is represented as

$$x_{\text{fix}} = \text{sign} \times (\alpha_n 2^n + \alpha_{n-1} 2^{n-1} + \cdots + \alpha_0 2^0 + \cdots + \alpha_{-m} 2^{-m}). \tag{2.4}$$

That is, one bit is used to store the sign and the remaining $N-1$ bits are used to store the α_i values ($n + m = N - 2$). The particular values for N, m, and n are machine-dependent. For a 32-bit machine, the integers are typically 4 bytes in length and in the range

$$-2147483648 \leq \text{integer} * 4 \leq 2147483647\,. \tag{2.5}$$

The advantage of the representation (2.4) is that you can count on all fixed-point numbers to have the same absolute error of 2^{-m-1} [the term left off the right-hand end of (2.4)]. The corresponding disadvantage is that *small* numbers (those for which the first string of α values are zeros) have large *relative* errors. Because in the real world relative errors tend to be more important than absolute ones, fixed-point numbers are used mainly in special applications (like business).

Your scientific work will mainly use floating-point numbers. In floating-point notation, the number x is stored as a sign, a mantissa, and an exponential field *expfld*. The number is reconstituted as

$$x_{\text{float}} = (-1)^s \times \text{mantissa} \times 2^{\text{expfld - bias}}. \tag{2.6}$$

Here the mantissa contains the significant figures of the number, s is the sign bit, and the actual exponent of the number has the *bias* added to it and is then stored as the exponential field *expfld*.

Just as introducing a sign bit guarantees that the mantissa is always positive, so introducing the bias guarantees that the number stored as the exponent field in (2.6) is always positive (the actual exponent of the number can, of course, be negative). The use of bias is rather indirect. For example, a single-precision 32-bit word may use 8 bits for the exponent in (2.6) and represent it as an integer. This 8-bit integer "exponent" has a range $[0, 255]$. Numbers with actual negative exponents are represented by a bias equal to 127, a fixed number for a given machine. Consequently, the exponent has the range $[-127, 128]$ even though the value stored for the exponent in (2.6) is a positive number. Of the remaining bits, one is used for the sign and 23 for the mantissa.

It is important to remember that single-precision (4-byte) numbers have 6–7 decimal places of precision (1 part in 2^{23}) and magnitudes typically in the range

$$10^{-44} \leq \text{single precision} \leq 10^{38}. \tag{2.7}$$

The mantissa of a floating number is represented in memory in the form

$$\text{mantissa} = m_1 \times 2^{-1} + m_2 \times 2^{-2} + \cdots + m_{23} \times 2^{-23}, \tag{2.8}$$

with just the m_i stored, similar to (2.4). As an example, the number 0.5 is stored as

```
0   0111 1111   1000 0000 0000 0000 0000 000,
```

where the bias is $0111\ 1111_2 = 127_{10}$.

In order to have the same relative precision for all floating-point numbers, it is standard to normalize the number so that the leftmost bit is unity, $m_1 = 1$. Once this convention is adopted, the m_1 does not even have to be stored and the computer only needs to recall that there is a *phantom bit*. During the processing of numbers in a calculation, the first bit of an intermediate result may become zero, but this will be corrected before the final number is stored.

Typically, the largest possible floating-point number for a 32-bit machine

```
0   1111 1111   1111 1111 1111 1111 1111 111
```

has the value 1 for all its bits (except sign) and adds up to $2^{128} = 3.4 \times 10^{38}$. Typically, the smallest possible floating-point number,

```
0   0000 0000   1000 0000 0000 0000 0000 000
```

has the value 0 for almost all its bits and adds up to $2^{-128} = 2.9 \times 10^{-39}$. As built in by the use of bias, the smallest number possible to store is the inverse of the largest.

If you write a program requesting *double precision*, then 64-bit (8-byte) words will be used in place of the 32-bit (4-byte) words. With 11 bits used for the exponent and 52 for the mantissa, double-precision numbers have about 16 decimal places of precision (1 part in 2^{52}) and typically have magnitudes

in the range

$$10^{-322} \leq \text{double precision} \leq 10^{308}. \tag{2.9}$$

2.13 IMPLEMENTATION: OVER- AND UNDERFLOWS, OVER.F (.C)

Write a program to test for the underflow and overflow limits (within a factor of 2 at least) of your computer system and of your favorite computer language. A sample pseudocode is

```
under = 1.
over = 1.
  begin do N times
    under = under/2.
    over = over * 2.
    write out:  loop number, under, over
  end do
```

You may need to increase N if your initial choice does not lead to underflow and overflow. Be careful to notice whether your computer's implementation of your programming language converts overflows, as well as underflows, to zero. (Converting underflows to zero is usually a good thing to do; converting overflows to zero is usually a good way to cause a disaster.) Notice that if you want to be more precise regarding the limits of your computer, you may want to multiply and divide by a number smaller than 2.

1. Check where under- and overflow occur for single-precision floating-point numbers.

2. Check where under- and overflow occur for double-precision floating-point numbers.

3. Check where under- and overflow occur for integers (you need to multiply and subtract 1 to see the effect).

2.14 MODEL: MACHINE PRECISION

One consequence of a computer's memory scheme for numbers is that the numbers can be recalled with only a limited precision. While the exact precision depends on the computer, single precision is usually 6–7 decimal places for a 32-bit word machine, and double precision is usually 15–16 places. (Some symbolic manipulation programs can store numbers with infinite precision; that is, the word size increases as the requisite precision increases.) To see how *machine precision* affects calculations, consider the simple addition of two 32-bit words:

$$7 \;+\; 1.0 \times 10^{-7} = ? \tag{2.10}$$

The computer fetches these numbers from memory and stores the bit patterns

$$\begin{aligned} 7 &= 0\ 10000010\ 1110\ 0000\ 0000\ 0000\ 0000\ 000, && (2.11)\\ 10^{-7} &= 0\ 01100000\ 1101\ 0110\ 1011\ 1111\ 1001\ 010, && (2.12) \end{aligned}$$

in *working registers* (pieces of fast-responding memory). Because the exponents are different, it would be incorrect to add the mantissas. So the exponent of the smaller number is made larger while progressively decreasing the mantissa by *shifting bits* to the right (inserting zeros) until both numbers have the same exponent:

$$\begin{aligned} 10^{-7} &= 0\ 01100001\ 0110\ 1011\ 0101\ 1111\ 1100101\ (0) \\ &= 0\ 01100010\ 0011\ 0101\ 1010\ 1111\ 1110010\ (10) && (2.13)\\ \cdots &, \\ &= 0\ 10000010\ 0000\ 0000\ 0000\ 0000\ 0000\ 000\ (0001101\cdots) \\ \Rightarrow & \ 7 \;+\; 1.0 \times 10^{-7} = 7 && (2.14) \end{aligned}$$

Because there is no more room left to store the last digits, they are lost. After all this hard work, the addition gives 7. This means a 32-bit computer only stores 6–7 decimal places and in effect ignores the 10^{-7}.

The preceding loss of precision is categorized by defining the *machine precision* ϵ_m as the maximum positive number that, on the computer, can be added to the number stored as 1 without changing the number stored as 1:

$$1_c + \epsilon_m = 1_c, \tag{2.15}$$

where the subscript c is a reminder that this is the number stored in the computer's memory. Likewise, x_c, the computer's representation of x, and the actual number x, are related by

$$x_c = x(1+\epsilon), \quad |\epsilon| \le \epsilon_m \,.$$

In other words, $\epsilon \simeq 10^{-7}$ for single precision and 10^{-16} for double precision.

If a single-precision number x is larger than 2^{128}, an *overflow* occurs. If x is smaller than 2^{-128}, an *underflow* occurs. The resulting number x_c may end up being a machine-dependent pattern, or *NAN* (not a number), or unpredictable. Because the only difference between the representations of positive and negative numbers on the computer is the sign bit of one for negative numbers, the same considerations hold for negative numbers.

In our experience, serious scientific calculations almost always require double precision, especially on 32-bit machines. And if you need double precision in one part of your calculation, you probably need it all over, and that also means double-precision library routines.

2.15 IMPLEMENTATION: DETERMINING YOUR PRECISION, LIMIT.F (.C)

Write a program to determine the machine precision ϵ of your computer system (within a factor of 2 at least). A sample pseudocode is

```
eps = 1.
  begin do N times
    eps = eps/2.                              Make smaller
    one = 1.  + eps
    write out:  loop number, one, eps
  end do
```

1. Check the precision for single-precision floating-point numbers.

2. Check the precision for double-precision floating-point numbers.

To print out a decimal number, the computer must make a conversion from its internal format. Not only does this take time, but if the internal number is close to being garbage, it's not clear what will get printed out. So if you want a truly precise indication of the stored numbers, you want to avoid conversion to decimals and, instead, print them out in octal or hexadecimal format.

2.16 PROBLEM 3: COMPLEX NUMBERS AND INVERSE FUNCTIONS

The language of physics is mathematics. Therefore using a computer to do physics ultimately means using it to do mathematics. But as we have seen, mathematics is not the native language of computers. The **problem** for you to investigate is the way your computer handles complex numbers and inverse trigonometric functions.

2.17 THEORY: COMPLEX NUMBERS

A complex number z is defined in terms of its real and imaginary parts as

$$z = x + iy. \tag{2.16}$$

It is also defined in terms of its magnitude r and phase ϕ as

$$\begin{aligned} z &= re^{i\phi}, && (2.17)\\ \text{where} \quad r &= \sqrt{x^2+y^2}, \quad \phi = \tan^{-1}\left(\frac{y}{x}\right). && (2.18) \end{aligned}$$

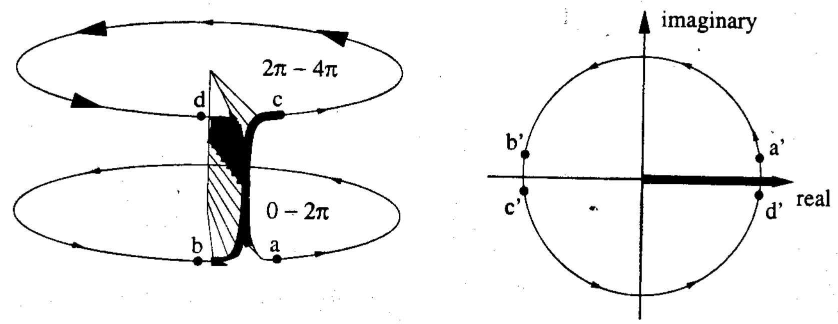


Fig. 2.8 (*Left*) The complex plane represented as two Riemann sheets attached through a cut appropriate to the $\sqrt{z}$ function. (*Right*) The cut complex plane. The cut prevents us from getting to a' from d' with a small rotation.

With this second definition we see that the square root and logarithms of z must be

$$\sqrt{z} = \sqrt{r}e^{i\phi/2}, \quad (2.19)$$
$$\ln z = \ln r + i\phi, \quad (2.20)$$

where $\ln r$ is the standard natural logarithm of a real number. That being so, while the phase ϕ can have an arbitrary multiple of 2π added to it *without* changing the location of z in the complex plane, this addition *does* change the values of $\sqrt{z}$ and $\ln z$. (Although adding 4π to the phase ϕ returns both z and $\sqrt{z}$ to the same complex plane location, the ln function never returns.)

A way of avoiding this apparent multivalueness of functions is to agree not to encircle the origin $z = 0$. To build this agreement into the mathematics, a *branch cut* is drawn along the intersection of sheets, in this case from the *branch point* $z = 0$ to $z = \infty$, as shown in Fig. 2.8. One then agrees not to pass through this branch cut. Typically the cut lies along the real x axis, or the imaginary y axis, although any line will do.

On the left in Fig. 2.8 we show a complex plane made up of two *Riemann sheets* attached through the cut in the shaded region. A rotation of 2π takes us from a to b but not back to a. After point b, the rotation moves to point c on a second Riemann sheet, then to d, and, after a total rotation of 4π, back down to the first sheet at a. On the right in Fig. 2.8 we show the conventional complex plane. Because there is a cut along the positive x axis (as occurs for the $\sqrt{z}$), the points a' and d' are not as close to each other as are the points c' and b'.

Once the complex plane is cut, we imagine crossing the cut, but without getting into multivalueness troubles, by passing onto another Riemann sheet that is joined to the original sheet along the cut. For a doubled-valued function like $\sqrt{z}$, passing through the cut twice, once from each sheet, returns us to our starting point. While this ambiguity may not be much of a problem for clever mathematicians, it is for operationally minded computers who only know to do what they are told to do. For you to uncover if this is a problem for your computer, or how some programmer has decided to resolve the ambiguity, you need to work out these exercises.

2.18 IMPLEMENTATION: COMPLEX NUMBERS, COMPLEX.C (.F)

Fortran is nice enough to do complex arithmetic and evaluate inverse functions for you. Some C compilers, like those from Borland and Linux, have extensions to handle complex numbers, but a standard C compiler does not. Problem-solving environments, like Maple and Mathematica, are usually also good with complex numbers.

Write a program that gets the computer to print a table of the form

ϕ	x	y	e^z	$\sqrt{z}$	$\ln z$	$\operatorname{atan}(y/x)$	$\operatorname{atan2}(y,x)$
-4π	*	*	*	*	*	*	*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
$15\pi/4$	*	*	*	*	*	*	*
4π	*	*	*	*	*	*	*

Here ϕ will increase with uniform steps, and the columns marked $\sqrt{z}$, log, atan, and atan2 are to be the computer's output. (If your compiler cannot handle complex numbers, forget about the z terms.)

1. Make a plot of the output phases obtained with the arctangent functions versus the input phase ϕ. [Notice that if $r = 1$ in (2.20), then the ln function is purely imaginary.]

2. If your plotting program appears to be making some strange jumps, you may need to use more points near a multiple of $\pi/2$ and avoid being precisely "at" a multiple of $\pi/2$.

3. If your compiler is not bright enough to automatically use a complex library routine when you feed it a complex number, you may have to look up the particular function name required to evaluate a complex function.

4. State clearly where the computer has placed the cuts for sqrt, ln, atan, and atan2 functions.

2.19 EXPLORATION: COMPLEX ENERGIES IN QUANTUM MECHANICS

A particle *not* under the influence of a potential is called "free." In quantum mechanics, a free particle moving in one dimension is described by the *plane wave*

$$\phi(x,t) = e^{i(kx-\omega t)}. \tag{2.21}$$

We obtain the probability density ρ for finding this particle at position x at time t by computing the squared modulus of the wave function $\rho = |\phi(x,t)|^2$. A trick used to describe an *unstable* system that *decays exponentially* in time is to say it can still be described by (2.21), only now with the complex ω (energy).

$$\hbar\omega = E_r - i\Gamma/2. \tag{2.22}$$

(Of course, if it decays, it cannot truly be "free," but that is why this is a model and not a theory.)

1. Show analytically that for positive or negative values of Γ, the probability ρ decays or grows (respectively) with increasing time t.

2. Show analytically that if energy and momentum are related in the usual way

$$\hbar\omega = \frac{k^2}{2m}, \tag{2.23}$$

then the momentum k also becomes a complex number, and the probability ρ then decays or grows with increasing distance x.

3. Write a program to calculate k for arbitrary values of E_r and Γ. Make the program *interactive*; that is, have it read E_r and Γ from the terminal and print *all* possible k values on the screen of the terminal.

4. Check the momentum values predicted by your program as both E_r and Γ change sign. Describe the physical reasonableness of its predictions.

2.20 PROBLEM 4: SUMMING SERIES

A classic numerical problem is the summation of a series to evaluate a function. For this exercise we examine the power series for the exponential function:

$$e^{-x} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \cdots \quad (x^2 < \infty). \tag{2.24}$$

We want to use the series (2.24) to calculate e^{-x} for $x = 0.1$, 1, 10, 100, and 1000, with an absolute error in each case of less than one part in 10^8. But how do we know when to stop summing? (Do not dare say or even think of saying, "When the answer agrees with the table or with the built-in library function.")

2.21 METHOD: NUMERIC

While we really want to ensure a definite accuracy for e^{-x}, that is not so easy to do. What is easy is to assume that the error in the summation is approximately the last term summed (this also assumes no roundoff error, which we will discuss soon). To obtain an absolute error of one part in 10^8, we can stop the calculation when

$$\left|\frac{\text{term}}{\text{sum}}\right| < 10^{-8}, \tag{2.25}$$

where *term* is the last term in the series (2.24), and *sum* is the accumulated sum.

2.22 IMPLEMENTATION: PSEUDOCODE

A pseudocode for performing the summation is

```
term = 1, sum = 1, eps = 10**(-8)              Initialize.
do
  term = -term * x/i                   New term in terms of old.
  sum = sum + term                                 Add in term.
  while abs(term/sum) > eps          Break iteration if accurate.
end do
```

You can see that this technique saves time by avoiding raising x to an ever-increasing power, and by never calculating the factorial.