2.23 IMPLEMENTATION: GOOD ALGORITHM, EXP-GOOD.F (.C)

Write a program that implements this pseudocode for the indicated x values. Present your results as a table of the form

x	imax	sum	$\frac{\lvert \text{sum} - \text{exp(-x)} \rvert}{\text{sum}}$

where `exp(-x)` is calculated with the built-in exponential function.

2.24 IMPLEMENTATION: BAD ALGORITHM, EXP-BAD.F (.C)

Modify your code that sums the series in a "good way" (no factorials) to one that calculates the sum in a "bad way" (explicit factorials). A sample is given by `exp-bad.f (.c)`.

2.25 ASSESSMENT

1. Observe how the good and bad series summations fail for large x. In particular, notice whether there are underflows or overflows.
2. Produce a table as above.
3. Use a built-in timing function on the computer to compare the time for each method.

3

Errors and Uncertainties in Computations

Whether you like it or not, errors and uncertainties are a part of computation. Some errors are the ones humans inevitably keep on making, but many are introduced by the computer. Computer errors arise either because of the limited *precision* with which a computer stores numbers or because sometimes it really does make mistakes (particularly in sophisticated chores like compilation with optimization). Although it stifles creativity to keep thinking *error* when approaching a computation, it certainly is a waste of time to generate meaningless results because of errors. In this chapter we examine some of the errors and *uncertainties* introduced by the computer.

3.1 PROBLEM: LIVING WITH ERRORS

Let's say you have a program of significant complexity. To gauge why errors are such a concern, let us assume that your program has the logical flow

$$\text{start} \rightarrow U_1 \rightarrow U_2 \rightarrow \ldots \rightarrow U_n \rightarrow \text{end}, \tag{3.1}$$

where each unit U might be a step. If each unit has probability p of being correct, then the probability P of the whole program being correct is $P = p^n$. Let's say we have a large program, say, with $n = 1000$ steps, and that the probability of each step being correct is $p = 0.9993$. This means that you end up with $P = \frac{1}{2}$; that is, a final answer that is as likely wrong as right (not what you want to bring to your boss). The **problem** is that, as a scientist, you want a result that is correct—or at least in which the uncertainty is small.

3.2 THEORY: TYPES OF ERRORS

Four general types of errors exist to plague your computations:

Blunders: typographical errors entered with your program or data, running the wrong program, using the wrong data file, and so on. (If your blunder count starts increasing, it is time to go home or take a break.)

Random errors: those caused by events such as fluctuation in electronics due to power surges, cosmic rays, or someone pulling a plug. These may be rare but you have no control over them and their likelihood increases with running time; while you may have confidence in a 20-second calculation, a week-long calculation may have to be run several times to check reproducibility.

Approximation errors: those arising from simplifying the mathematics so that a problem can be solved or approximated on the computer. They include the replacement of: infinite series by finite sums, infinitesimal intervals by finite ones, and variable functions by constants. For example

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} \tag{3.2}$$

$$\simeq \sum_{n=0}^{N} \frac{x^n}{n!} = e^x + \mathcal{E}(x, N), \tag{3.3}$$

where $\mathcal{E}(x, N)$ is the total absolute error. Because approximation error arises from the application of the mathematics, it is also called *algorithmic error*, the *remainder*, or *truncation error*.[1] The approximation error clearly decreases as N increases, and vanishes in the $N \to \infty$ limit. Specifically for (3.3), because the scale for N is set by the value of x, small approximation error requires $N \gg x$. So if x and N are close in value, the approximation error will be large.

Roundoff errors: those arising very much like the uncertainty in the measurement of a physical quantity encountered in an elementary physics laboratory. Because any stored number is represented by a finite number of bits (and consequently digits), the set of numbers that the computer can store exactly, *machine numbers*, is much smaller than the set of real numbers. In particular, there is a maximum and minimum to machine numbers. The overall error arising from using a finite number of digits

[1]The use of the term "truncation" may be somewhat confusing here because it is sometimes also used to describe the truncation of digits in the representation of a number, an effect more usually referred to as *roundoff error*.

to represent numbers accumulates as the computer handles more numbers; that is, as the number of steps in a computation increases. In fact, roundoff error causes some algorithms to become *unstable* with a rapid increase in error for certain parameters. In some cases, roundoff error may exceed the number itself, leaving what computer experts call *garbage*. For example, as computed (and you may try this at home)

$$2\left(\tfrac{1}{3}\right) - \tfrac{2}{3} = 0.6666666 - 0.6666667 = -0.0000001 \neq 0. \tag{3.4}$$

When dealing with roundoff error, you may be sensitive as to whether this error arises from "subtractive cancellation" or "multiplicative cancellation." And when considering these cancellations, it is good to recall those discussions of *significant figures* and scientific notation given in your early physics or engineering classes. For computational purposes let us consider how the computer may store the floating-point number

$$a = 11223344556677889900 = 1.12233445566778899 \times 10^{19}. \tag{3.5}$$

Because the exponent is stored separately and is a small number, we can assume that it will be stored in full precision. The mantissa may not be stored completely, depending on the word length of the computer and whether we declare the word to be stored in single or double precision. In double precision (or *REAL*8* on a 32-bit machine or *doubles*), the mantissa of a will be stored as two words, the *most significant part* representing the decimal 1.12233, and the *least significant part* 44556677. The digits beyond 7 may be lost. As we see below, when we perform calculations with words of fixed length, it is inevitable that errors get introduced into the least significant parts of the words.

3.3 MODEL: SUBTRACTIVE CANCELLATION

An operation performed on a computer usually only approximates the analytic answer. The approximation arises because computers are finite. Let us use the notation in which the number x is represented on the computer as x_c. The representation of a simple subtraction is then

$$a = b - c \quad \Rightarrow \quad a_c = b_c - c_c, \tag{3.6}$$

$$a_c = b(1 + \epsilon_b) - c(1 + \epsilon_c), \tag{3.7}$$

$$\Rightarrow \quad \frac{a_c}{a} = 1 + \epsilon_b \frac{b}{a} - \frac{c}{a}\epsilon_c. \tag{3.8}$$

We see from (3.8) that, in the crudest sense, the average error in a is a weighted average of the errors in b and c. Yet there can also be exceptional cases. The error in a increases when $b \approx c$ because we subtract off (and thereby lose) the most significant parts of both numbers. This leaves the least significant parts. This is a general rule:

> *If you subtract two large numbers and end up with a small one, there will be less significance in the small one.*

In other words, if a is small it must mean $b \simeq c$ and so

$$\frac{a_c}{a} = 1 + \epsilon_a, \tag{3.9}$$

$$\epsilon_a \simeq \frac{b}{a}(\epsilon_b - \epsilon_c). \tag{3.10}$$

This shows that even if the relative errors in b and c cancel somewhat, they are multiplied by the large number b/a, which, in turn, can make a differ significantly from a_c, even for small ϵ. If the signs of the numbers are such that the magnitude of a turns out to be larger than those of either b or c, this means that the numbers have been added together, in which case there is no subtractive cancellation and we can expect an accurate representation.

A good example of subtractive cancellation occurs in the power series summation for e^{-x} studied in Chapter 2, *Computing Software Basics*. For very large x, the early terms in the series can be quite large, but because the final answer must be very small, most of the large terms must be cancelled out. Consequently, one approach is to calculate e^x for very large x, and then take its inverse to obtain e^{-x}. This eliminates the subtractive cancellation occurring between successive terms because all terms in e^x just add.

3.4 ASSESSMENT: SUBTRACTIVE CANCELLATION EXPERIMENT

1. Remember back in high school when you learned that the quadratic equation

$$ax^2 + bx + c = 0, \tag{3.11}$$

has the analytic solution

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}, \tag{3.12}$$

or alternatively

$$x'_{1,2} = \frac{-2c}{b \pm \sqrt{b^2 - 4ac}}. \tag{3.13}$$

Inspection of (3.12)–(3.13) indicates that subtractive cancellation (and consequently an increase in relative error) arises when $b^2 \gg 4ac$ because then the square root and its preceding term nearly cancel. If $b > 0$, this subtractive cancellation occurs in x_1 and x'_2, while for $b < 0$ it occurs in x'_1 and x_2.

(a) Write a program that calculates all four solutions for arbitrary values of a, b, and c.

(b) Investigate how errors in your computed answers become large as the subtractive cancellation increases, and relate this to the known machine precision. (*Hint*: A good test case employs $a = 1$, $b = 1$, $c = 10^{-n}$, $n = 1, 2, 3, \ldots$.)

(c) Extend your program so that it will always tell you which are the most precise solutions.

2. You also have to be careful to avoid subtractive cancellation when summing a series. For example, consider the finite sum with alternating signs:

$$S_N^{(1)} = \sum_{n=1}^{2N} (-1)^n \frac{n}{n+1}. \tag{3.14}$$

If you sum the even and odd values of n separately, you get two sums

$$S_N^{(2)} = -\sum_{n=1}^{N} \frac{2n-1}{2n} + \sum_{n=1}^{N} \frac{2n}{2n+1}. \tag{3.15}$$

All terms are positive in this form with just a single subtraction at the end of the calculation. Even this one subtraction and its resulting cancellation can be avoided by combining the series analytically:

$$S_N^{(3)} = \sum_{n=1}^{N} \frac{1}{2n(2n+1)}. \tag{3.16}$$

While all three summations are mathematically equal, this may not be true numerically.

(a) Write a single-precision program that calculates $S^{(1)}, S^{(2)}$, and $S^{(3)}$.

(b) Assume $S^{(3)}$ to be the exact answer. Make a log–log plot of the relative error versus number of terms, that is, of $\log_{10}|(S_N^{(1)} - S_N^{(3)})/S_N^{(3)}|$, versus $\log_{10}(N)$. Start with $N = 1$ and work up to $N = 1,000,000$.

(c) See whether straight-line behavior occurs in some region of your plot.

3. In spite of the power of your trusty computer, calculating the sum of even a simple series may require some thought and care. Consider the series

$$S^{(\text{up})} = \sum_{n=1}^{N} \frac{1}{n}, \tag{3.17}$$

which is finite as long as N is finite. When summed analytically, it does not matter if you sum the series upward from $n = 1$ or downward from $n = N$,

$$S^{(\text{down})} = \sum_{n=N}^{1} \frac{1}{n}. \tag{3.18}$$

Nonetheless, because of roundoff error, when summed numerically, $S^{(\text{up})} \neq S^{(\text{down})}$.

(a) Write a program to calculate $S^{(\text{up})}$ and $S^{(\text{down})}$ as functions of N.

(b) Make a log–log plot of the relative difference divided by the relative sum versus N.

(c) Observe the linear regime on your graph and explain why the downward sum is more precise.

3.5 MODEL: MULTIPLICATIVE ERRORS

Error in computer multiplication arises in the following way:

$$a = b \times c \quad \Rightarrow \quad a_c = b_c \times c_c, \tag{3.19}$$

$$\Rightarrow \quad \frac{a_c}{a} = \frac{(1+\epsilon_b)(1+\epsilon_c)}{(1+\epsilon_a)} \simeq 1 + \epsilon_b + \epsilon_c. \tag{3.20}$$

Since ϵ_b and ϵ_c can have opposite signs, the error in a_c is sometimes larger and sometimes smaller than the individual errors in b_c and c_c.

It often turns out that we can estimate an average roundoff error for a series of multiplications by assuming that the computer's representation of a number differs *randomly* from the actual number. In these situations we have the analog of a random walk (which is discussed in Chapter 6, *Deterministic Randomness*). If the direction of each step in the walk is random, then R, the average distance covered in N steps each of length r, is

$$R \approx \sqrt{N} r. \tag{3.21}$$

Equation (3.20) indicates that each step of a multiplication has a roundoff error of length ϵ_m, the machine precision. Imagine making physical steps of length ϵ_m. By analogy to a random walk, the average relative error ϵ_{ro} arising after a large number N steps is

$$\epsilon_{ro} \approx \sqrt{N} \epsilon_m. \tag{3.22}$$

We will find (3.22) useful when we examine the error in algorithms.

For those situations in which the roundoff errors do not occur in a random manner, a careful analysis is needed to predict the dependence of the error on the number of steps N. In some cases there may be no cancellation of error and the relative error may well increase like $N\epsilon_m$. Even worse, in some recursive algorithms where the production of errors is coherent (e.g., upward recursion for Bessel functions), the error increases like $N!\epsilon_m$.

Our discussion of errors has an important implication for a student to keep in mind before being impressed by a calculation requiring hours of supercomputer time. A fast computer may complete 10^{10} floating-point operations per second. This means a program running for 3 hours performs about 10^{14} operations. Therefore, even in the best case of random errors, after 3 hours we expect roundoff errors to have accumulated to a relative importance of $10^7\epsilon_m$. For the error to be smaller than the answer, this demands $\epsilon_m < 10^{-7}$. As a result, we can make the generalization that the results of a several-hours-long calculation with 32-bit arithmetic (which inherently possesses only six to seven places of precision) probably contains much noise. This fact is seldom appreciated by users of large amounts of computer time.

3.6 PROBLEM 1: ERRORS IN SPHERICAL BESSEL FUNCTIONS

Accumulating roundoff errors often limits the ability of a program to perform accurate calculations. Your **problem** is the computation of the spherical Bessel and Neumann functions j_l and n_l.

Spherical Bessel functions occur in many physical problems, for example; the j_l's are part of the partial wave expansion of a plane wave into spherical waves,

$$e^{i\mathbf{k}\cdot\mathbf{r}} = \sum_{l=0}^{\infty} i^l (2l+1) j_l(kr) P_l(\cos\theta), \tag{3.23}$$

where θ is the angle between $\mathbf{k}$ and $\mathbf{r}$. Fig. 3.1 shows what a number of j_l's look like, and Table 3.1 gives some explicit values. The spherical Bessel function $j_l(x)$ is the solution of the differential equation

$$x^2 f''(x) + 2x f'(x) + \left[x^2 - l(l+1)\right] f(x) = 0, \tag{3.24}$$

which is regular (nonsingular) at the origin. The spherical Neumann function $n_l(x)$ is a second, independent solution of (3.24). It is irregular (diverges at $x = 0$), and is chosen to contain just the right amount of j_l needed for proper asymptotic behavior. Specifically

$$\begin{array}{llll} j_l(x) & \rightarrow & x^l/(2l+1)!! & \text{for } x \ll l, \\ n_l(x) & \rightarrow & -(2l-1)!!/x^{l+1} & \text{for } x \ll l, \\ j_l(x) & \sim & \sin(x - l\pi/2)/x & \text{for } x \gg l, \\ n_l(x) & \sim & -\cos(x - l\pi/2)/x & \text{for } x \gg l, \\ \text{where } (2l+1)!! & \equiv & 1\cdot 3\cdot 5 \cdots (2l+1). & \end{array} \tag{3.25}$$

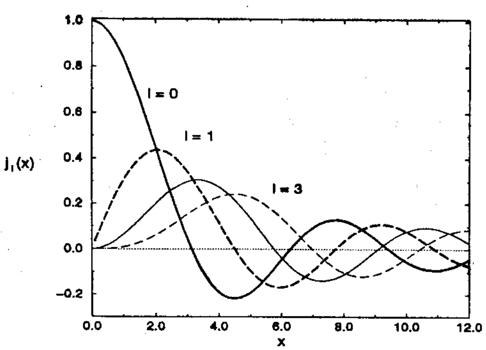

Fig. 3.1 The first four spherical Bessel functions, $j_l(x)$, as functions of x.

3.7 METHOD: NUMERIC RECURSION RELATIONS

One way to write a computer program to calculate $j_l(x)$ is to deduce its power series and asymptotic expansion. You then use these to evaluate $j_l(x)$ for small and large x/l, respectively [possibly augmented by a direct integration of the differential equation (3.24) for values in between]. The needed equations can be found in [Jack 75] and [A&S 64].

The approach we investigate here is often quicker than the use of series and has the advantage of generating the spherical Bessel functions for *all* l values at one time (for fixed x). It is based on the *recursion relation*:

$$j_{l+1}(x) = \frac{2l+1}{x} j_l(x) - j_{l-1}(x), \quad \text{(up)}, \tag{3.26}$$

$$j_{l-1}(x) = \frac{2l+1}{x} j_l(x) - j_{l+1}(x), \quad \text{(down)}. \tag{3.27}$$

Equations (3.26) and (3.27) both express the same relation, one written for recurring upward and the other for recurring downward. With just a few additions and multiplications, this recurrence relation permits a rapid and simple computation of the entire set of j_l's for fixed x and all l.

To recur upward we start with the fixed value of x and the known forms for j_0 and j_1:

$$j_0(x) = \frac{\sin(x)}{x}, \quad j_1(x) = \frac{\sin(x) - x\cos(x)}{x^2}. \tag{3.28}$$

We then use (3.26) to calculate $j_l(x)$ for all higher l values.

As you yourself will see, this upward recurrence usually starts working

Table 3.1 Approximate values for spherical Bessel functions of orders 3, 5, and 8 at $x = 0.1$, 1.0, and 10

x	$j_3(x)$	$j_5(x)$	$j_8(x)$
0.10	$+9.518510^{-6}$	$+9.616310^{-10}$	$+2.901210^{-16}$
1.00	$+9.006610^{-3}$	$+9.256110^{-05}$	$+2.826510^{-08}$
10.0	-3.949610^{-2}	-5.553510^{-02}	$+1.255810^{-01}$

pretty well but then fails. The reason for the failure can be seen from the plots of $j_l(x)$ and $n_l(x)$ versus x. If we start at $x \approx 2$ and $l = 0$, then we see from the graph that as we recur j_l up to larger l values with the relation (3.26), we are essentially taking the difference of two "large" numbers to produce a "small" one. This always reduces the precision. As we continue recurring, we are taking the difference of two "small" numbers to produce a smaller number yet, and this increases the relative error. After a while, the ever-increasing subtractive cancellations mean we are left with only roundoff error (garbage).

In contrast, if we use the upward recurrence relation (3.26) to produce the spherical Neumann function n_l, there is no problem. In that case, the graph makes clear that we are combining small numbers to produce larger ones, and in this way do not have any subtractive cancellation. In that case we are always working with the most significant parts of the numbers.

To be more specific, let us call $j_l^{(c)}$ the numerical value we compute as an approximation for $j_l(x)$. Even if we start with pure j_l, after a short while the computer's lack of precision effectively mixes in a bit of $n_l(x)$:

$$j_l^{(c)} = j_l(x) + \epsilon n_l(x). \tag{3.29}$$

This is inevitable because both j_l and n_l satisfy the same differential equation, and on that account, the same recurrence relation.

The admixture of n_l becomes a problem if the numerical value of n_l is much larger than that of j_l, because then even a miniscule amount of a very large number may be large. We can see from the limits (3.25) that if $l \gg x$, then the Neumann function is larger, $n_l > j_l$. This means that the error behaves like the spherical Neumann function, and consequently grows without bounds at the origin for upward recurrence.

The simple solution to this problem is *Miller's device*: use (3.27) for downward recursion starting at a large value of l. This essentially takes two small j values and produces a larger one by addition and in this way avoids subtractive cancellation. While the error may still behave like a Neumann function, the actual magnitude of the error will *decrease* quickly as we move downward to smaller l values. In fact, we start iterating downward with arbitrary values (garbage) for $j_{N+1}^{(c)}$ and $j_N^{(c)}$, and after a short while we arrive at very good answers. While the numerical value of $j_0^{(c)}$ so obtained will not be correct because it depends on the explicit value assumed for "garbage," the ratio of the $j_l^{(c)}$ values will be accurate. Therefore, after you have finished the downward recurrence, you use the analytic expression for $j_0^{(c)}$ (3.28) to normalize $j_0^{(c)}$ and all higher $j_l^{(c)}$ values.

3.8 IMPLEMENTATION: RECURSION RELATIONS, BESSEL.F (.C)

1. Write a program to calculate $j_l(x)$ that will give "good" values for the first 25 l values for $x = 0.1, 1.0, 10.0$ ["good" means a relative error $\simeq 10^{-6}$ (10^{-14}) for single (double) precision]. See Table 3.1 for some sample values.
2. Try it with both upward and downward recursion, but don't try too hard for upward recursion. (Try using single precision in order to see error effects more quickly.)

3.9 ASSESSMENT

1. Give results of the downward recursion for different, large values of the starting l, showing the convergence and stability of your results.
2. Compare the upward and downward recursion methods, printing out l, $j_l^{(\mathrm{up})}$, $j_l^{(\mathrm{down})}$, and the relative difference

$$\frac{|j_l^{(\mathrm{up})} - j_l^{(\mathrm{down})}|}{|j_l^{(\mathrm{up})}| + |j_l^{(\mathrm{down})}|}. \tag{3.30}$$

3. The errors in the upward recursion depend on x, and for certain values of x, both up and down recursions give similar answers. Explain the reason for this and what it tells you about your program.

3.10 PROBLEM 2: ERRORS IN ALGORITHMS

Numerical algorithms play a vital role in computational physics. You start with a physical theory or mathematical model, you use algorithms to convert the mathematics into a calculational scheme, and, finally, you convert your scheme into a computer program. Your **problem** is to take a general algorithm, and decide

1. Does it converge?
2. How precise are the results when it does converge?
3. How expensive (time consuming) is it to run ?

3.11 MODEL: ERRORS IN ALGORITHMS

An algorithm is often characterized by its step size h or by the number of steps N it takes to reach its goal. If the algorithm is "good," it should give the exact answer in the limit $h \to 0$ or $N \to \infty$. Every algorithm contains an *approximation error*; that is, there is a difference between the exact result and the result of the algorithm. If you know the approximation error as a function of the number of terms N used in the approximation, you may be able to judge "when enough is enough already." Yet do not be misled into believing the "error" has vanished because you made N so ridiculously large that the approximation error must be small. The *total* error in your calculation also includes roundoff errors, systematic errors, and possibly bad input data, all of which tend to increase when you make the computer work harder.

In general, as you continue to decrease the step size h or increase the number of steps N, you will reach a point where the roundoff error has grown large enough to exceed your approximation error. Clearly, the optimum choice of your parameters are those that minimize the total error. Unfortunately, in many cases there is no simple expression to minimize. Yet by using some of the methods described here, you may be able to determine the behavior of your error and so gain some control over it.

3.11.1 Total Error

Let us assume that an algorithm takes a large number N steps to get a good answer and that the approximation error approaches zero like

$$\epsilon_{\mathrm{aprx}} \simeq \frac{\alpha}{N^\beta}. \tag{3.31}$$

Here α and β are empirical constants that would change for different algorithms, and may be "constant" only for $N \to \infty$. As indicated at the beginning of this chapter, the roundoff error keeps accumulating as you take more steps; that is, it increases with N. If the roundoff errors in the individual steps of your algorithm are not correlated, then we know from our previous discussion that

$$\epsilon_{\mathrm{ro}} \simeq \sqrt{N}\epsilon_m, \tag{3.32}$$

where ϵ_m is the machine precision. The total error would be the sum of the two:

$$\begin{aligned}\epsilon_{\text{tot}} &= \epsilon_{\text{aprx}} + \epsilon_{\text{ro}}, &(3.33)\\ &\simeq \frac{\alpha}{N^\beta} + \sqrt{N}\epsilon_m. &(3.34)\end{aligned}$$

Although no discussion of errors is exciting (except maybe for masochists), it is useful. We assume we have a test case for which a good answer is known either analytically or from some other source. By comparing the test case answers to those computed, we deduce the total error ϵ_{tot} in the calculation. If we then plot $\log(\epsilon_{\text{tot}})$ against $\log(N)$, we can use the slope of this graph [the power of N in the expansion of the error (3.34)] to deduce which error term is dominant for differing N values. Alternatively, by starting at very large N values where we expect there to be essentially no approximation error, we can move in to smaller values of N and thereby deduce the N behavior of the approximation error.

If you run your test case with N much smaller than α/N^β, then the approximation error term in (3.34) should dominate and the slope should be $-\beta$. If N is much larger than $\sqrt{N}\epsilon_m$, then the roundoff error term should dominate and the slope should be $\frac{1}{2}$. If your test case does not have this behavior, there may be a problem in your program, or the model may be too simple.

3.12 METHOD: OPTIMIZING WITH KNOWN ERROR BEHAVIOR

In order to see more clearly how different kinds of errors balance off each other, let us now turn to the *relative* size of errors. We will assume the approximation error (3.31) has $\alpha = 1, \beta = 2$:

$$\epsilon_{\text{aprx}} \simeq \frac{1}{N^2}. \qquad (3.35)$$

If the total error is given by (3.34), then it will have an extremum when

$$\frac{d\epsilon_{tot}}{dN} = 0 \quad \Rightarrow \quad N^{\frac{5}{2}} = \frac{4}{\epsilon_m}. \qquad (3.36)$$

Because a maximum total error occurs for $N = \infty$, the extremum should be a minimum. For a computer with 32-bit words and single precision, $\epsilon_m \simeq 10^{-7}$, so the minimum total error (3.36) occurs when

$$\begin{aligned} N^{\frac{5}{2}} \simeq \frac{4}{10^{-7}} \quad &\Rightarrow \quad N \simeq 1099, &(3.37)\\ \epsilon_{\text{tot}} \quad &\simeq \quad \frac{1}{N^2} + \sqrt{N}\epsilon_m &(3.38)\end{aligned}$$

$$= \quad 8 \times 10^{-7} + 33 \times 10^{-7} \simeq 4 \times 10^{-6}. \qquad (3.39)$$

This shows that for a typical algorithm, most of the error is due to roundoff. Observe, too, that even though this is the minimum error, the best we can do is to get some 40 times machine precision (the double-precision results are better).

Seeing that total error is mainly roundoff error, an obvious way to decrease the total error is to decrease roundoff error by using a smaller number of steps N. Let us assume we do this by finding another algorithm that converges more rapidly with N, for example, one for which the approximation error behaves like

$$\epsilon_{\text{aprx}} \simeq \frac{2}{N^4}. \qquad (3.40)$$

The total error is now

$$\epsilon_{\text{tot}} = \frac{2}{N^4} + \sqrt{N}\epsilon_m. \qquad (3.41)$$

The number of points for minimum error is found as before

$$\begin{aligned} \frac{d\epsilon_{tot}}{dN} = 0 \quad &\Rightarrow \quad N^{\frac{9}{2}} = \frac{16}{\epsilon_m}, &(3.42)\\ \epsilon_m \simeq 10^{-7} \quad &\Rightarrow \quad N \simeq 67, &(3.43)\\ \epsilon_{\text{tot}} \quad &= \quad \frac{2}{N^4} + \sqrt{N}\epsilon_m &(3.44)\\ &= \quad 1 \times 10^{-7} + 8 \times 10^{-7} \simeq 9 \times 10^{-7}. &(3.45)\end{aligned}$$

The error is now smaller by a factor of 4 with only $\frac{1}{16}$ as many steps needed. Subtle are the ways of the computer. In this case it is not that the better algorithm is more elegant, but rather, by being quicker and using fewer steps, it produces less roundoff error.

3.13 METHOD: EMPIRICAL ERROR ANALYSIS

Let us say you have a program you want to optimize for minimum total error, yet you do not know (or do not want to go to the trouble of deriving) what the approximation error is. As just discussed, you know that in some approximate and general sense, the roundoff error ϵ_{ro} is related to the machine precision ϵ_m and the number of calculational steps N by

$$\epsilon_{ro} \simeq \sqrt{N}\epsilon_m. \qquad (3.46)$$

Because the approximation error should get smaller with larger N, the roundoff error ϵ_{ro} should dominate the total error for very large N.

Let us assume that the exact answer to your problem is $\mathcal{A}$, while that obtained by your algorithm after N steps is $A(N)$. The trick then is to examine the behavior of $A(N)$ for values of N large enough for the approximation error to have its asymptotic value (the term with the smallest inverse power of N dominates), but not too large to dominate roundoff error. In this case we can write

$$A(N) \simeq \mathcal{A} + \frac{\alpha}{N^\beta}, \tag{3.47}$$

where α and β are unknown constants. We now run our computer program with a large number N of steps, and again with twice that number of steps. If roundoff error is not yet dominating, then

$$A(N) - A(2N) \approx \frac{\alpha}{N^\beta}. \tag{3.48}$$

To actually see if these assumptions are correct, and see graphically the number of decimal places to which the solution has converged, you plot $\log_{10}|A(N)/A(2N) - 1|$ versus $\log_{10} N$. That part of your plot that is a straight line indicates the region in which the assumptions are valid, and the slope gives the value for $-\beta$.

If N is too small, we would not be in the asymptotic region for the approximation error, and the graph will not be a straight line. As N gets much larger, roundoff error begins to enter and the graph departs from the previous straight line (it should change slope to something like $+\frac{1}{2}$).

All this means that you can figure out what is happening with your algorithm by experimenting: start off with small N and increase it until you get a reasonably straight-line graph. Then increase N and watch as the graph changes slope to a positive one appropriate to roundoff error. Because the ordinate is the logarithm of the relative error to the base 10, it immediately tells you the number of decimal places of precision obtained.

3.14 ASSESSMENT: EXPERIMENT

Consider the series for the exponential function

$$e^{-x} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \cdots \quad (x^2 < \infty), \tag{3.49}$$

$$\simeq \sum_{n=0}^{N} \frac{(-x)^n}{n!}. \tag{3.50}$$

To most readily see the effects of error accumulation in this algorithm, use single precision for your programming.

1. Write a program that calculates e^{-x} as the finite sum (3.50).
2. Try $x = 1, 10,$ and 100.

3. Examine the terms in the series for $x \simeq 10$ and observe the significant subtractive cancellations that occur when large terms add together to give small answers. See if better precision is obtained by using $\exp(-x) = 1/\exp(x)$ for large x values.
4. By progressively increasing N, use your program to experimentally determine whether there is a range of N for which the approximation error is asymptotic and yet larger than roundoff error. (You may assume that the built-in exponential function is exact.)
5. Determine whether (3.47) is valid and, if so, determine the values for β.

Because this series summation is such a simple and correlated process, the roundoff error does not accumulate randomly as it might for a more complicated computation, and we do not obtain the error behavior (3.47). To really see this error behavior, try this test with the integration rules discussed in Chapter 4, *Integration*.