3
Errors and Uncertainties in Computations

Whether you are careful or not, errors and uncertainties are always a part of computation. Some errors are the ones that humans inevitably make, but some are introduced by the computer. Computer errors arise because of the limited precision *with which computers store numbers, or because programs can get things wrong. Although it stifles creativity to keep thinking* "error" *when approaching a computation, it certainly is a waste of time to work with results that are meaningless (*"garbage"*) because of errors. In this chapter we examine some of the errors and* uncertainties *introduced by the computer.*

3.1
Living with Errors (Problem)

Let us say you have a program of significant complexity. To gauge why errors may be a concern for you, imagine a program with the logical flow:

$$\text{start} \quad \rightarrow U_1 \rightarrow U_2 \rightarrow \cdots \rightarrow U_n \rightarrow \quad \text{end} \tag{3.1}$$

where each unit U might be a step. If each unit has probability p of being correct, then the joint probability P of the whole program being correct is $P = p^n$. Let us say we have a large program 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 a good way to do science). 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
Types of Errors (Theory)

Four general types of errors exist to plague your computations:

Blunders or bad theory: Typographical errors entered with your program or data, running the wrong program or having a fault in your reasoning

Computationyal Physics. Problem Solving with Computers (2nd edn).
Rubin H. Landau, Manuel José Páez, Cristian C. Bordeianu
Copyright © 2007 WILEY-VCH Verlag GmbH & Co. KGaA, Weinheim
ISBN: 978-3-527-40626-5

(theory), using the wrong data file, and so on. (If your blunder count starts increasing, it may be time to go home or take a break.)

Random errors: Errors 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: Errors 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,

$$e^{-x} = \sum_{n=0}^{\infty} \frac{(-x)^n}{n!} \simeq \sum_{n=0}^{N} \frac{(-x)^n}{n!} = e^{-x} + \mathcal{E}(x, N) \tag{3.2}$$

where $\mathcal{E}(x, N)$ is the approximation error. Because the approximation error arises from the application of the mathematics, it is also called *algorithmic error*. The approximation error clearly decreases as N increases, and vanishes in the $N \to \infty$ limit. Specifically for (3.2), because the scale for N is set by the value of x, the small approximation error requires $N \gg x$. So if x and N are close in value, the approximation error will be large.

Roundoff errors: Imprecisions arising from the finite number of digits used to store floating-point numbers. These "errors" are analogous to the uncertainty in the measurement of a physical quantity encountered in an elementary physics laboratory. The overall error arising from using a finite number of digits to represent numbers accumulates as the computer handles more numbers, that is, as the number of steps in a computation increases. In fact, the roundoff error causes some algorithms to become *unstable* with a rapid increase in error for certain parameters. In some cases, the roundoff error may become the major component in your answer, leading to what computer experts call *garbage*. For example, if your computer kept four decimal, then it would store 1/3 as 0.3333 and 2/3 as 0.6667, where the computer has "rounded off" the last digit in 2/3. Accordingly, if we ask the computer to as simple a calculation as $2(1/3) - 2/3$, it produces

$$2\left(\tfrac{1}{3}\right) - \tfrac{2}{3} = 0.6666 - 0.6667 = -0.0001 \neq 0 \tag{3.3}$$

So even though the result is small, it is not 0, and if we repeat this type of calculation millions of times, the answer might not even be small.

When considering the precision of calculations, 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:

$$a = 11223344556677889900 = 1.12233445566778899 \times 10^{19} \tag{3.4}$$

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 `double`), the mantissa of a will be stored in 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 for Disaster: Subtractive Cancellation

An operation performed on a computer usually only approximates the analytic answer because the numbers are stored only approximately. To demonstrate the effect of this type of uncertainty, let us call x_c the computer representation of the exact number x. The two are related by

$$x_c \simeq x(1 + \epsilon_x) \tag{3.5}$$

where ϵ_x is the relative error in x_c, which we expect to be of a similar magnitude to the machine precision ϵ_m. If we apply this notation to the simple subtraction $a = b - c$, we obtain

$$\begin{aligned} a = b - c \qquad &\Rightarrow \qquad a_c \simeq b_c - c_c \simeq b(1+\epsilon_b) - c(1+\epsilon_c) \\ &\Rightarrow \qquad \frac{a_c}{a} \simeq 1 + \epsilon_b \frac{b}{a} - \frac{c}{a}\epsilon_c \end{aligned} \tag{3.6}$$

We see from (3.6) that the resulting error in a is essentially a weighted average of the errors in b and c, with no assurance that the terms will cancel. Of special importance here is to observe that the error in the answer a increases when we subtract two nearly equal numbers ($b \approx c$), because then we are subtracting off the most significant parts of both numbers and leaving the error-prone least significant parts:

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

In terms of our error analysis, if the answer from your subtraction a is small, it must mean $b \simeq c$ and so

$$\frac{a_c}{a} \overset{\text{def}}{=} 1 + \epsilon_a \simeq 1 + \frac{b}{a}(\epsilon_b - \epsilon_c) \simeq 1 + \frac{b}{a}\max(|\epsilon_b|, |\epsilon_c|) \tag{3.7}$$

This shows that even if the relative errors in b and c may cancel somewhat, they are multiplied by the large number b/a, which can significantly magnify the error. Because we cannot assume any sign for the errors, we must assume the worst (the "max" in (3.7)).

We have already seen an example of subtractive cancellation in the power series summation $e^{-x} \simeq 1 - x + x^2/2 + \cdots$ in Chap. 3. For large x, the early terms in the series are large, but because the final answer must be very small, these large terms must subtract each other away to give the small answer. Consequently, a better approach for calculating e^x eliminates subtractive cancellation by calculating e^x, and then setting $e^{-x} = 1/e^x$. As you should check for yourself, this improves the algorithm, but does not make it good because there is still a roundoff error.

3.4 Subtractive Cancellation Exercises

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

$$ax^2 + bx + c = 0 \tag{3.8}$$

has an analytic solution that can be written as either

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \quad \text{or} \quad x'_{1,2} = \frac{-2c}{b \pm \sqrt{b^2 - 4ac}} \tag{3.9}$$

Inspection of (3.9) indicates that subtractive cancellation (and consequently an increase in error) arises when $b^2 \gg 4ac$ because then the square root and its preceding term nearly cancel for one of the roots.

   (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 indicates the most precise solutions.

2. As we have seen, subtractive cancellation occurs when summing a series with alternating signs. As another example, consider the finite sum:

$$S_N^{(1)} = \sum_{n=1}^{2N} (-1)^n \frac{n}{n+1} \tag{3.10}$$

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

$$S_N^{(2)} = -\sum_{n=1}^{N} \frac{2n-1}{2n} + \sum_{n=1}^{N} \frac{2n}{2n+1} \tag{3.11}$$

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:

$$S_N^{(3)} = \sum_{n=1}^{N} \frac{1}{2n(2n+1)} \tag{3.12}$$

While all three summations $S^{(1)}, S^{(2)}$, and $S^{(3)}$ 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 $N = 1,000,000$. (Recall, $\log_{10} x = \ln x/\ln 10$.)

(c) See whether straight-line behavior for the error occurs in some region of your plot. This indicates that the error is proportional to some power of N.

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 two series

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

Both series are finite as long as N is finite, and when summed analytically, it does not matter if you sum the series upward or downward. Nonetheless, because of the roundoff error, the numerical $S^{(\text{up})} \neq S^{(\text{down})}$.

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

(b) Make a log–log plot of $(S^{(\text{up})} - S^{(\text{down})})/(|S^{(\text{up})}| + |S^{(\text{down})}|)$ divided by the sum versus N.

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

3.5 Model for Roundoff Error Accumulation

Let us start by seeing how error arises in a single multiplication of the computer representation of two numbers:

$$\begin{aligned} a = b \times c \qquad &\Rightarrow \qquad a_c = b_c \times c_c \ = b(1+\epsilon_b) \times c(1+\epsilon_c) \\ &\Rightarrow \qquad \frac{a_c}{a} = (1+\epsilon_b)(1+\epsilon_c) \simeq 1 + \epsilon_b + \epsilon_c \end{aligned} \tag{3.13}$$

where we ignore very small ϵ^2 terms.[1] This is just the basic rule of error propagation from elementary physics or engineering laboratory: you add the uncertainties in each quantity involved in an analysis in order to determine the overall uncertainty. As before, the safest assumption is that there is no cancellation of error, that is, that the errors add in absolute value.

There is a useful model for approximating how the roundoff error accumulates in a calculation involving a large number of steps. We view the error in each step as a literal "step" in a *random walk*, that is, a walk for which each step is in a random direction. As we derive and simulate in Chap. 11 the total distance covered in N steps of length r, is, on average,

$$R \approx \sqrt{N}\, r \tag{3.14}$$

By analogy, the total relative error ϵ_{ro} arising after N calculational steps each with the machine precision error ϵ_m, is, on average,

$$\epsilon_{\mathrm{ro}} \approx \sqrt{N} \tag{3.15}$$

If the roundoff errors in a particular algorithm do not accumulate in a random manner, then a detailed analysis is needed to predict the dependence of the error on the number of steps N. In some cases there may be no cancellation and the error may increase like $N\epsilon_m$. Even worse, in some recursive algorithms, where the production of errors is coherent, such as the upward recursion for Bessel functions, the error increases like $N!$.

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 h performs approximately 10^{14} operations. Therefore, if the roundoff error accumulates randomly, after 3 h we expect a relative error of $10^7\epsilon_m$. For the error to be smaller than the answer, we need $\epsilon_m < 10^{-7}$, which means double-precision calculations.

1 We thank B. Gollsneider for informing us of an error in an earlier version of this equation.

3.6 Errors in Spherical Bessel Functions (Problem)

Accumulating roundoff errors often limits the ability of a program to perform accurate calculations. Your **problem** is to compute the spherical Bessel and Neumann functions $j_l(x)$ and $n_l(x)$. These are, respectively, the regular (non-singular at the origin) and irregular solutions of the differential equation

$$x^2 f''(x) + 2x f'(x) + \left[x^2 - l(l+1)\right] f(x) = 0 \tag{3.16}$$

and are related to the Bessel function of the first kind by $j_l(x) = \sqrt{\pi/2x} J_{n+1/2}(x)$. 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,

$$e^{i\mathbf{k}\cdot\mathbf{r}} = \sum_{l=0}^{\infty} i^l\,(2l+1) j_l(kr)\, P_l(\cos\theta) \tag{3.17}$$

where θ is the angle between **k** and **r**. Figure 3.1 shows what the first few j_l's look like, and Tab. 3.1 gives some explicit values. For the first two l values, explicit forms are

$$j_0(x) = +\frac{\sin x}{x} \qquad j_1(x) = +\frac{\sin x}{x^2} - \frac{\cos x}{x} \tag{3.18}$$

$$n_0(x) = -\frac{\cos x}{x} \qquad n_1(x) = -\frac{\cos x}{x^2} - \frac{\sin x}{x} \tag{3.19}$$

3.7 Numeric Recursion Relations (Method)

The classic way to calculate $j_l(x)$ would be by summing its power series for small values of x/l, and its asymptotic expansion for large values. The approach we adopt here is quicker 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 relations*:

$$j_{l+1}(x) = \frac{2l+1}{x} j_l(x) - j_{l-1}(x) \qquad \text{(up)} \tag{3.20}$$

$$j_{l-1}(x) = \frac{2l+1}{x} j_l(x) - j_{l+1}(x) \qquad \text{(down)} \tag{3.21}$$

Equations (3.20) and (3.21) are the same relation, one written for upward recurrence from small to large l, and the other for downward recurrence from large to small l. With just a few additions and multiplications, a recurrence relation permits a rapid and simple computation of the entire set of j_l's for fixed x and all l.

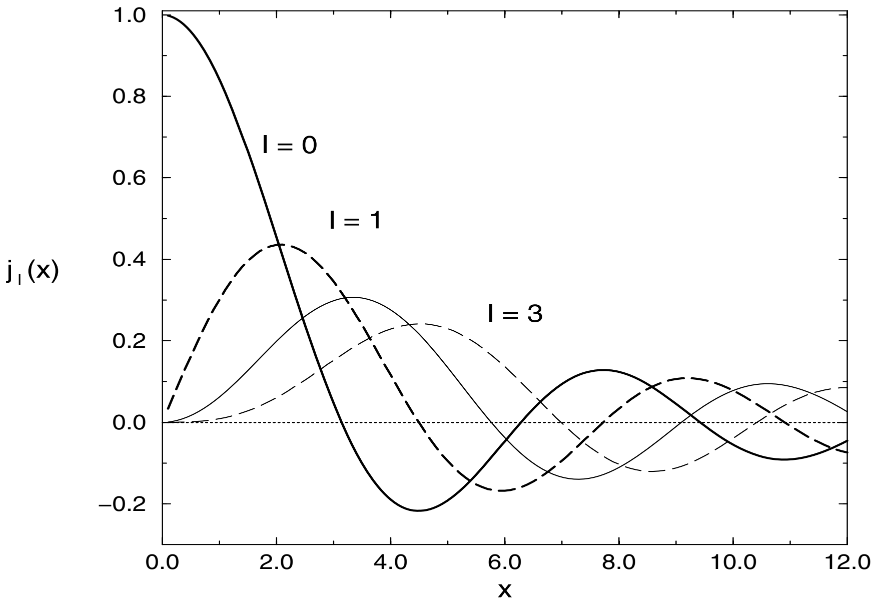

Fig. 3.1 The first four spherical Bessel functions $j_l(x)$ as functions of x. Notice how for small x, the values for increasing l become progressively smaller.

To recur upward for fixed x, we start with the known forms for j_0 and j_1, (3.18), and use (3.20). As you yourself will see, this upward recurrence usually starts working 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 (Fig. 3.1). If we start at $x \approx 2$ and $l = 0$, then we see that as we recur j_l up to larger l values with (3.20), we are essentially taking the difference of two "large" numbers to produce a "small" one. This process suffers from subtractive and always reduces the precision. As we continue recurring, we take the difference of two "small" numbers with large errors and produce a smaller number with yet larger relative error. After a while, we are left with only roundoff error (garbage).

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 $n_l(x)$:

$$j_l^{(c)} = j_l(x) + \epsilon n_l(x) \tag{3.22}$$

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 $n_l(x)$ is much larger than that of $j_l(x)$. Then even a minuscule amount of a very large number may be large.

Tab. 3.1 Approximate values for spherical Bessel functions of orders 3, 5, and 8 (from Maple).

x	$j_3(x)$	$j_5(x)$	$j_8(x)$
0.1	$+9.518519719 \times 10^{-6}$	$+9.616310231 \times 10^{-10}$	$+2.901200102 \times 10^{-16}$
1	$+9.006581118 \times 10^{-3}$	$+9.256115862 \times 10^{-05}$	$+2.826498802 \times 10^{-08}$
10	$-3.949584498 \times 10^{-1}$	$-5.553451162 \times 10^{-01}$	$+1.255780236 \times 10^{+00}$

In contrast, if we use the upward recurrence relation (3.20) to produce the spherical Neumann function n_l, there is no problem. In that case we are combining small numbers to produce larger ones (Fig. 3.1), a process that does not contain subtractive cancellation, and so we are always working with the most significant parts of the numbers.

The simple solution to this problem (*Miller's device*) is to use (3.21) for downward recursion starting at a large value of l. This avoids subtractive cancellation by taking small values of $j_{l+1}(x)$ and $j_l(x)$ and producing a larger $j_{l-1}(x)$ by addition. 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, if we start iterating downward with arbitrary values for $j^{(c)}_{L+1}$ and $j^{(c)}_{L}$, and after a short while we arrive at the correct l dependence for this value of x. Although the numerical value of $j^{(c)}_0$ so obtained will not be correct because it depends upon the arbitrary values assumed for $j^{(c)}_{L+1}$ and $j^{(c)}_{L}$, we know from the analytic form (3.18) what the value of $j_0(x)$ should be. In addition, because the recurrence relation is a linear relation between the j_l's, we need only normalize all the computed values via

$$j_l^{\text{normalized}}(x) = j_l^{\text{compute}}(x) \times \frac{j_0^{\text{analytic}}(x)}{j_0^{\text{compute}}(x)} \tag{3.23}$$

Accordingly, after you have finished the downward recurrence, you normalize all $j^{(c)}_l$ values.

3.8 Implementation and Assessment: Recursion Relations

A program implementing recurrence relations is most easily written using subscripts. If you need to polish up your skills with subscripts, you may want to study our program `Bessel.java` in Listing 3.1 before writing your own.

Listing 3.1: The code `Bessel.java` that determines spherical Bessel functions by downward recursion (you should modify this to also work by upward recursion). Note that the comments (beginning with //) are on the right and are ignored by the compiler.

```java
// Bessel.java: Spherical Bessels via up and down recursion

import java.io.*;

public class bessel {
                                        // Global class variables
  public static double xmax = 40., xmin = 0.25, step = 0.1;
  public static int order = 10, start = 50;

  public static void main(String[] argv)
                       throws IOException, FileNotFoundException {

   double x;

   PrintWriter w =
        new PrintWriter(new FileOutputStream("Bessel.dat"), true);
                                               // Step thru x values
   for ( x = xmin;  x <= xmax;  x += step )
        w.println(" " +x+"  "+down(x, order, start) );
    System.out.println("data stored in Bessel.dat");
   }                                                     // End main

  public static double down (double x, int n, int m) {  // Recur down
   double scale;
   double j[] = new double[start + 2];
   int k;
                                               // Start with anything
   j[m + 1] = j[m] = 1.;
   for ( k = m;  k>0 ;  k--)   j[k-1] = ((2.*k+1.)/x)*j[k] - j[k+1];
                                       // Scale solution to known j[0]
   scale = (Math.sin(x)/x)/j[0];
   return j[n] * scale;
 }
}
```

1. Write a program that uses both upward and downward recursion to calculate $j_l(x)$ for the first 25 l values for $x = 0.1, 1, 10$.
2. Tune your program so that at least one method gives "good" values ("good" means a relative error $\simeq 10^{-10}$). See Tab. 3.1 for some sample values.
3. Show the convergence and stability of your results.
4. Compare the upward and downward recursion methods, printing out l, $j_l^{(\text{up})}$, $j_l^{(\text{down})}$, and the relative difference $|j_l^{(\text{up})} - j_l^{(\text{down})}|/|j_l^{(\text{up})}| + |j_l^{(\text{down})}|$.
5. The errors in computation 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.

3.9 Experimental Error Determination (Problem)

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 converged results?
3. How expensive (time consuming) is it?

3.10 Errors in Algorithms (Model)

On first thought you may think "What a dumb problem! All algorithms converge if enough terms are used, and if you want more precision then just use more terms." Well, some algorithms may be asymptotic expansions that just approximate a function in certain regions of parameter space, and only converge up to a point. Yet even if a uniformly convergent power series is used as the algorithm, including more terms will decreases the algorithmic error but increase the roundoff errors. And because the roundoff errors eventually diverge to infinity, the best we can hope for is a "best" approximation. "Good" algorithms are good because they yield an acceptable approximation in a small number of steps, thereby not giving the roundoff error time to grow large.

Let us assume that an algorithm takes N steps to get a good answer. As a rule of thumb, the approximation (algorithmic) error decreases rapidly, often as the inverse power of the number of terms used:

$$\epsilon_{\text{aprx}} \simeq \frac{\alpha}{N^{\beta}} \tag{3.24}$$

Here α and β are empirical constants that would change for different algorithms, and may be only approximately constant as $N \to \infty$. The fact that the error must fall off for large N is just a statement that the algorithm converges. In contrast to this algorithmic error, roundoff errors tend to grow slowly and somewhat randomly with N. If the roundoff errors in the individual steps of the algorithm are not correlated, then we know from our previous discussion that we can model the accumulation of error as a random walk with step size equal to the machine precision ϵ_m,

$$\epsilon_{\text{ro}} \simeq \sqrt{N}\epsilon_m \tag{3.25}$$

Indeed this is the slow growth with N we expect from the roundoff error. The total error in a computation would be the sum of the two:

$$\epsilon_{\rm tot} = \epsilon_{\rm aprx} + \epsilon_{\rm ro} \simeq \frac{\alpha}{N^{\beta}} + \sqrt{N}\epsilon_m \tag{3.26}$$

We expect the first term to be the larger of the two for small N, but ultimately to be overcome by the slowly growing second term. As an example, in Fig. 5.4 we present a log–log plot of the relative error in numerical integration using the Simpson integration rule (Chap. 5).

We use the $\log_{10}$ of the relative error because its negative tells us the number of decimal places of precision obtained.[2] As a case in point, let us assume that $\mathcal{A}$ is the exact answer and $A(N)$ the computed answer. If

$$\frac{\mathcal{A} - A(N)}{\mathcal{A}} = 10^{-9} \quad \text{then} \quad \log_{10}\left|\frac{\mathcal{A} - A(N)}{\mathcal{A}}\right| = -9 \tag{3.27}$$

We see in Fig. 5.4 that the error does show a rapid decrease for small N, consistent with an inverse power law (3.24). In this region the algorithm is converging. Yet as N keeps increasing, the error starts looking somewhat erratic, yet with a slow increase on average. In accordance with (3.26), in this region the roundoff error has grown larger than the approximation error and will continue to grow for increasing N. Clearly then, the smallest total error would be obtained if we can stop the calculation at the minimum near 10^{-14}, that is, when $\epsilon_{\rm aprx} \simeq \epsilon_{\rm ro}$.

In realistic calculation you would not know the exact answer; after all, if you did, then why would you bother with the computation? However, you may know the exact answer for a similar calculation, and you can use that similar calculation to perfect your numerical technique. Alternatively, now that you understand how the total error in a computation behaves, you should be able to look at a table or, better yet, graph (Fig. 5.4) of your answer, and deduce the manner in which your algorithm is converging. Specifically, at some point you should see that your answer begins to change only in the digits several places to the right of the decimal point, with that place moving further to the right as your calculation executes more steps. Eventually, however, as the number of steps gets truly large, the roundoff error should lead to some fluctuation in some high decimal place, and then to a decrease in stability of your answer. You should quit the calculation before this occurs.

Based upon the previous reasoning, another approach is to 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 the computed

[2] Note that most computer languages set their log function to $\ln = \log_e$. Yet since $x \overset{\text{def}}{=} a^{\log_a x}$, the conversion to base 10 is simply $\log_{10} x = \ln x / \ln 10$.

$A(N)$ for values of N large enough for the approximation to be converging according to (3.24), that is, so that

$$A(N) \simeq \mathcal{A} + \frac{\alpha}{N^\beta} \tag{3.28}$$

but not that large so that the roundoff error term in (3.26) dominates. We then run our computer program with $2N$ steps, which should give a better answer still. If the roundoff error is not yet dominating, then we can eliminate the unknown $\mathcal{A}$ by subtraction:

$$A(N) - A(2N) \approx \frac{\alpha}{N^\beta} \tag{3.29}$$

To see if these assumptions are correct, and determine what level of precision is possible for the best choice of N, you would plot $\log_{10} |(A(N) - A(2N)/A(2N)|$ versus $\log_{10} N$, similar to what we have done in Fig. 5.4. If you obtain a rapid straight-line drop off, then you know you are in the region of convergence and can deduce a value for β from the slope. As N gets larger, you should see the graph change from the previous straight line decrease to a slow increase as the roundoff error begins to dominate. Before this is a good place to quit. In any case, now you are in control of your error.

3.11 Minimizing the Error (Method)

In order to see more clearly how different kinds of errors enter into a computation, let us examine a case where we know α and β, for example,

$$\epsilon_{\text{aprx}} \simeq \frac{1}{N^2} \qquad \Rightarrow \qquad \epsilon_{\text{tot}} \simeq \sqrt{N}\epsilon_m \tag{3.30}$$

where the total error is the sum of roundoff and approximation errors. This total error is a minimum when

$$\frac{d\epsilon_{tot}}{dN} = 0 \;\Rightarrow\; N^{\frac{5}{2}} = \frac{4}{\epsilon_m} \tag{3.31}$$

For a single-precision calculation ($\epsilon_m \simeq 10^{-7}$), the minimum total error occurs when

$$N^{\frac{5}{2}} \simeq \frac{4}{10^{-7}} \qquad \Rightarrow \qquad N \simeq 1099 \qquad \Rightarrow \qquad \epsilon_{\text{tot}} \simeq 4 \times 10^{-6} \tag{3.32}$$

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 the total error is mainly the roundoff error $\propto \sqrt{N}$, an obvious way to decrease the error is to use a smaller number of steps N. Let us assume that we do this by finding another algorithm that converges more rapidly with N, for example, one with the approximation error behaving like

$$\epsilon_{\text{aprx}} \simeq \frac{2}{N^4} \tag{3.33}$$

The total error is now

$$\epsilon_{\text{tot}} = \epsilon_{\text{RO}} + \epsilon_{\text{apprx}} \simeq \frac{2}{N^4} + \sqrt{N}\epsilon_m \tag{3.34}$$

The number of points for the minimum error is found as before,

$$\frac{d\epsilon_{\text{tot}}}{dN} = 0 \quad \Rightarrow \quad N^{\frac{9}{2}} \quad \Rightarrow \quad N \simeq 67 \quad \Rightarrow \quad \epsilon_{\text{tot}} \simeq 9 \times 10^{-7} \tag{3.35}$$

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.

Exercise: Repeat the preceding error estimates for double precision. □

3.12 Error Assessment

As you have seen in the previous chapter, the mathematical definition of the exponential function e^{-x} is

$$e^{-x} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \cdots = \sum_{n=0}^{\infty} \frac{(-x)^n}{n!} \tag{3.36}$$

As long as $x^2 < \infty$, this series converges in the mathematical sense. Accordingly, a first attempt at an algorithm to compute the exponential might be

$$e^{-x} \simeq \sum_{n=0}^{N} \frac{(-x)^n}{n!} \tag{3.37}$$

Clearly for (3.37) to be a good algorithm, we need to have the first ignored term, $(-x)^{N+1}/(N+1)!$, to be small in comparison to the sum we are computing. This is expected to occur for x values at which the numerical summation is converging to an answer. In addition, we also need to have the *sum* of all the ignored terms to be small compared to the sum we are keeping, and the sum we are keeping to be a good approximation to the mathematical sum.

While, in principle, it should be faster to see the effects of error accumulation in this algorithm by using single-precision numbers (floats) in your programming, C and Java tend to use double-precision mathematical libraries, and so it is hard to do a pure single-precision computation. Accordingly, do these exercises in double precision, as you should for all scientific calculations involving floating-point numbers.

1. Write a program that calculates e^{-x} as the finite sum (3.37). (If you have done this in the previous chapter, then you may reuse that program and its results.)

2. Calculate your series for $x \leq 1$ and compare it to the built-in function `exp(x)` (you may assume that the built-in exponential function is exact). You should pick an N for which the next term in the series is no more than 10^{-7} of the sum up to that point,

$$\frac{|(-x)^{N+1}|}{(N+1)!} \leq 10^{-7} \left| \sum_{n=0}^{N} \frac{(-x)^n}{n!} \right| \tag{3.38}$$

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. In particular, print out the near-perfect cancellation at $n \simeq x - 1$.

4. See if better precision is obtained by being clever and using $\exp(-x) = 1/\exp(x)$ for large x values. This eliminates subtractive cancellation, but does not eliminate all roundoff errors.

5. By progressively increasing x from 1 to 10, and then from 10 to 100, use your program to determine experimentally when the series starts to lose accuracy, and when the series no longer converges.

6. Make a series of graphs of the error versus N for different values of x.

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 (3.28). To really see this error behavior, try this test with the integration rules discussed in Chap. 5.