

## CHAPTER 1

# A First Numerical Problem

Many problems encountered in physics involve ordinary differential equations. Examples include projectile motion, harmonic motion, and celestial mechanics, topics we will be discussing extensively in the next few chapters. We therefore begin with a problem involving a first-order differential equation and use it to introduce some computational techniques that will be employed extensively in later chapters. We will also proceed step by step through the construction of a program to deal with this problem, so as to illustrate in detail how a numerical approach is translated into a (working) computer program.

In this chapter it is not possible to provide a complete introduction to programming for students who have no previous exposure to the subject. Rather, our goal is to enable students with some (even limited) experience in programming to begin writing programs to treat the physics that will be encountered in this book. However, those students with no prior experience should not give up hope! With some extra effort and access to a good instructor or book on computer programming (or both), such students should be able to handle the material in this and later chapters.

### 1.1 RADIOACTIVE DECAY

It is well known that many nuclei are unstable. A typical example is the nuclear isotope  $^{235}\text{U}$  (the uranium nucleus that contains 143 neutrons and 92 protons, for a total of 235 nucleons), which has a small, but not insignificant, probability for decaying into two nuclei of approximately half its size, along with an assortment of protons, neutrons, electrons, and alpha particles. This process of radioactive decay is random in the following sense. If you were given a single  $^{235}\text{U}$  nucleus, you would not be able to predict precisely when its decay would take place. The best you could do would be to give the *probability* for decay. An equivalent way to describe such a process would be to give the average time for decay; for  $^{235}\text{U}$  the mean lifetime is approximately  $1 \times 10^9$  years.

It is useful to imagine that we have a sample containing a large number of  $^{235}\text{U}$  nuclei, which would usually be the case if we were actually doing an experiment to study radioactive decay. If  $N_U(t)$  is the number of uranium nuclei that are present in the sample at time  $t$ , the behavior is governed by the differential equation

$$\frac{dN_U}{dt} = -\frac{N_U}{\tau}, \quad (1.1)$$

where  $\tau$  is the “time constant” for the decay. You can show by direct substitution that the solution to this differential equation is

$$N_U = N_U(0) e^{-t/\tau}, \quad (1.2)$$

where  $N_U(0)$  is the number of nuclei present at  $t = 0$ . This solution may be familiar to you; similar equations and similar solutions are found in many other contexts.<sup>1</sup> We note that at time  $t = \tau$  a fraction  $e^{-1}$  of the nuclei that were initially present has not yet decayed. It turns out that  $\tau$  is also the mean lifetime of a nucleus.

## 1.2 A NUMERICAL APPROACH

While the differential equation (1.1) can be solved without resorting to a numerical approach, this problem is useful for introducing several computational methods that will be used extensively in later chapters. With that in mind we now consider a simple method for solving this problem numerically. Our goal is to obtain  $N_U$  as a function of  $t$ . Given the value of  $N_U$  at one particular value of  $t$  (usually at  $t = 0$ ), we want to estimate its value at later times. This is called an initial value problem, and various general approaches for solving such ordinary differential equations are discussed in Appendix A. Here we will describe one particularly useful line of attack that is based on the Taylor expansion for  $N_U$ ,

$$N_U(\Delta t) = N_U(0) + \frac{dN_U}{dt} \Delta t + \frac{1}{2} \frac{d^2 N_U}{dt^2} (\Delta t)^2 + \cdots, \quad (1.3)$$

where  $N_U(0)$  is the value of our function at time  $t = 0$ ,  $N_U(\Delta t)$  is its value at  $t = \Delta t$ , and the derivatives are evaluated at  $t = 0$ . If we take  $\Delta t$  to be small, then it is usually a good approximation to simply ignore the terms that involve second and higher powers of  $\Delta t$ , leaving us with

$$N_U(\Delta t) \approx N_U(0) + \frac{dN_U}{dt} \Delta t. \quad (1.4)$$

The same result can be obtained from the definition of a derivative. The derivative of  $N_U$  evaluated at time  $t$  can be written as

$$\frac{dN_U}{dt} \equiv \lim_{\Delta t \rightarrow 0} \frac{N_U(t + \Delta t) - N_U(t)}{\Delta t} \approx \frac{N_U(t + \Delta t) - N_U(t)}{\Delta t}, \quad (1.5)$$

where in the last approximation we have assumed that  $\Delta t$  is small but nonzero. We can rearrange this to obtain

$$N_U(t + \Delta t) \approx N_U(t) + \frac{dN_U}{dt} \Delta t, \quad (1.6)$$

which is equivalent to (1.4). It is important to recognize that this is an *approximation*, which is why it contains the  $\approx$  symbol, not the  $=$  symbol. The error terms that were dropped in deriving this result are of order  $(\Delta t)^2$ , which makes them at least one factor of  $\Delta t$  smaller than any of the terms in (1.6). Hence, by making  $\Delta t$  small, we would expect that the error terms can be made negligible. This is, in fact, the case in many problems, but there are situations in which the error terms can

<sup>1</sup>For example, an equation of this kind describes the time dependence of the voltage across a capacitor in an  $RC$  circuit.

still make life complicated. Therefore, it is important to be careful when discussing the errors involved in this numerical approach; we will return to this point later in this chapter, and in more detail in Appendix A.

From the physics of the problem we know the functional form of the derivative (1.1), and if we insert it into (1.6) we obtain

$$N_U(t + \Delta t) \approx N_U(t) - \frac{N_U(t)}{\tau} \Delta t. \quad (1.7)$$

This approximation forms the basis for a numerical solution of our radioactive decay problem. Given that we know the value of  $N_U$  at some value of  $t$ , we can use (1.7) to *estimate* its value a time  $\Delta t$  later.<sup>2</sup> Usually we are given, or can manage to discover, the initial value of the function, that is, the value at time  $t = 0$ . We can then employ (1.7) to estimate its value at  $t = \Delta t$ . This result can be used in turn to estimate the value at  $t = 2\Delta t$ ,  $3\Delta t$ , etc., and thereby lead to an approximate solution  $N_U(n\Delta t)$  at times  $n\Delta t$  where  $n$  is an integer.<sup>3</sup> We cannot emphasize too strongly that the numerical “solution” obtained in this way is only an *approximation* to the “true,” or exact, solution. Of course, one of our goals is to make the difference between the two negligible.

The approach to calculating  $N_U(t)$  embodied in (1.6) and (1.7) is known as the *Euler method* and is a useful general algorithm for solving ordinary differential equations. We will use this approach, and closely related methods, extensively in this book. Other methods for solving equations of this kind will be discussed in later chapters, and more systematic discussions of all of these approaches and the typical errors associated with them are the subject of Appendix A. For now, the reader should realize that while the Euler method arises in a very natural way, it is certainly not the only algorithm for dealing with problems of this sort. We will see that the different approaches have their own strengths and weaknesses, which make them more or less suitable for different types of problems.

## 1.3 DESIGN AND CONSTRUCTION OF A WORKING PROGRAM: CODES AND PSEUDOCODES

In the previous section we introduced the Euler method as the basis for obtaining a numerical solution to our radioactive decay problem. We now consider how to translate that algorithm into a working computer program. Perhaps the first choice that one must make in writing a program is the choice of programming language. From the authors' experiences, there are many programming languages that are well suited for the kinds of problems we address in this book, and it is impossible for us to give example programs in all of these languages. However, it is possible to describe the structure of a program in a general way that is useful to users of many different languages. We will do this using a “language” known as *pseudocode*. This is not a precise programming language, but rather a description of the essential

<sup>2</sup>As you might expect, the quality of this estimate, i.e., its *accuracy*, will depend on the value of  $\Delta t$ . This is a very important issue that we will be discussing in some detail below and in Appendix A.

<sup>3</sup>Note that errors made each at each time step, i.e., each time (1.7) is used, will accumulate.

parts of an algorithm, expressed in “common” language. The idea is to give enough detail so that you (the readers of this book) can see how to translate each piece of pseudocode into the specific instructions of your favorite programming language. In most of this book, we will give our examples only in pseudocode. However, in this chapter we will work through an example using pseudocode along with actual codes in two popular languages, Fortran and C, so that you can see how the translation from pseudocode to an actual programming language can be done.<sup>4</sup> Working programs for many of the problems in this book are available in Fortran, C, and Basic at our Web site.<sup>5</sup>

While programming, like handwriting, is a highly individualized process, there are certain recommended practices. After all, as in handwriting, it is important that we be able to understand programs written by others, as well as those we ourselves have written! With that in mind, this book will try to promote proper programming habits. The (admittedly very loose) analogy between handwriting and programming can be carried one step further. The first thing you should do in writing any program is to *think*. Before writing any detailed code, construct an outline of how the problem is to be solved and what variables or parameters will be needed. Indeed, the pseudocode version of a program will often provide this outline. For our decay problem we have already laid the foundation for a numerical solution in our derivation of (1.7). This equation also contains all of the variables we will need,  $N_U$ ,  $t$ ,  $\tau$ , and  $\Delta t$ . Our stated goal was to calculate  $N_U(t)$ , but since the numerical approximation (1.7) involves the values of  $N_U$  only at times  $t = 0$ ,  $t = \Delta t$ ,  $t = 2\Delta t$ , etc., we will actually calculate  $N_U$  at just these values of  $t$ . We will use an array to store the values of  $N_U$  for later use. An array is simply a table of numbers (which will be described in more detail shortly). The first element in our array, that is, the first entry in the table, will contain  $N_U$  at  $t = 0$ , the second element will be the value at  $t = \Delta t$ , and so on. Our general plan is then to apply (1.7) repetitively to calculate the values of  $N_U(t)$ .

The overall structure of the program consists of four basic tasks: (1) declare the necessary variables, (2) initialize all variables and parameters, (3) do the calculation, and (4) store the results.

#### EXAMPLE 1.1 Pseudocode for the main program portion of the radioactive decay problem

- Some comment text to describe the nature of the program.
  - ▷ Declare necessary variables and arrays.
  - ▷ *initialize* variables.
  - ▷ Do the actual *calculation*.
  - ▷ *store* the results.

<sup>4</sup>We are certainly not implying that everyone should use Fortran or C, but these are the authors' favorites.

<sup>5</sup>[www.physics.purdue.edu/~giordano/comp-phys.html](http://www.physics.purdue.edu/~giordano/comp-phys.html)

Note that this is only the *main program*; the individual tasks such as initialization of variables and the actual calculation, will be done in subroutines or functions, which we will discuss below. First we consider how this main program might look in Fortran.<sup>6</sup>

---

#### radioactive decay main program in Fortran

---

```
c Simulation of radioactive decay
c Program to accompany "Computational Physics" by N. Giordano/H. Nakanishi
  program decay
c   declare the arrays we will need
    double precision n_uranium(100), t(100)
c use subroutines to do the work
    call initialize(n_uranium,t,tau,dt,n)
    call calculate(n_uranium,t,tau,dt,n)
    call store(n_uranium,t,n)
    stop
  end
```

---

Our program begins with a few comment statements that identify the program and tell a little about what it is supposed to do. In Fortran, comment lines are indicated by the ‘c’ in the first character in a line. Similar features are present in most languages.

The first line `program decay` gives the name of the main program. The line `double precision n_uranium(100), t(100)` declares the two arrays that will be used to store  $N_U$  and  $t$ . This is done in the main program because the subroutines that do the work will pass the arrays among each other through the main program. The first array, `n_uranium()`, will contain the calculated values of the number of uranium nuclei, while `t()` will contain the corresponding values of the time. Here we have arranged for our arrays `n_uranium()` and `t()` to each hold 100 values, as we expect this to be enough for this problem (of course, many languages allow one to resize an array as needed during a calculation). Note that we have used the descriptive name `n_uranium` to make the program easier to read and understand. It is also tempting to use the name `time` for the other array (instead of `t`), but some languages use `time` as a “reserved name” (e.g., the name of the time-of-day function), so to be safe we will avoid using it.

The rest of the work is done in three subroutines, `initialize`, `calculate`, and `store` which are called in succession. The subroutine names describe the function of each; these tasks correspond directly to the general program outline mentioned above. The `call` statements also include the names of the variables that each routine needs to do its job. The subroutine `initialize` sets the initial values of the variables, `calculate` uses the Euler method to do the computation,

<sup>6</sup>In the example Fortran codes that we list in this chapter, we have chosen an old (some may say *ancient*) style for widest compatibility. Today's Fortrans (*Fortran90* and *Fortran95*), though backward compatible with these examples, allow codes to be written in much more robust and extensible ways. However, we are not teaching a computer language in this book, and thus have decided to stick with the style that is most compatible with whatever compiler the reader may have.

and `store` puts the results into a file for later use (such as a graphical display). We next consider these three routines.

### EXAMPLE 1.2 Pseudocode for subroutine initialize

- Prompt for and assign  $N_U(0)$ ,  $\tau$ , and  $\Delta t$ .
- Set initial value of time,  $t(0)$ .
- Set number of time steps for calculation.

A Fortran version of this subroutine could read

---

Fortran version of subroutine initialize

---

```

c      subroutine initialize(nuclei,t,tc,dt,n)
c      Initialize variables
      double precision nuclei(1),t(1)
      print *, 'initial number of nuclei -> '
      read(5,*) nuclei(1)
      print *, 'time constant -> '
      read(5,*) tc
      print *, 'time step -> '
      read(5,*) dt
      print *, 'total time -> '
      read(5,*) time
      t(1) = 0
      n=min(int(time/dt),100)
      return
      end

```

---

In Fortran all subroutines begin with the `subroutine` name statement where name is the subroutine name. As in most other languages, the variables listed in parenthesis after the word `initialize` are passed into and out of the subroutine from/to the calling routine. This variable list must be in correspondence with the list used when the subroutine is called from the main program (or anywhere else). Comparing the calling line and the first line of the subroutine, we see that some of the names in the variable list in the “calling” part of the program, `n_uranium`, `t`, `tau`, `dt`, `n`, and the list in the “receiving” part of the program, `nuclei`, `t`, `tc`, `dt`, `n` are quite different. For example, the array names `n_uranium` and `nuclei` do not agree. But, as in most languages, the array and variable names declared within a subroutine definition are *dummy* names, and only the corresponding arrays and variables in the calling program are actually used.

After getting the calling variables organized, the `initialize` subroutine sets the initial values of the number of nuclei and the time. These are just the first values in the arrays `nuclei(1)` and `t(1)`. The `print` statements prompt the user for input, while the `read` statements take in the values. The last job is to set the value of `n`, which is the number of time steps to be performed.

The real work of computing the number of remaining nuclei is done in the subroutine `calculate`.

### EXAMPLE 1.3 Pseudocode for subroutine calculate

- For each time step  $i$  (beginning with  $i = 1$ ), calculate  $N_U$  and  $t$  at step  $i + 1$ :
  - ▷  $N_U(t_{i+1}) = N_U(t_i) - (N_U(t_i)/\tau)dt$  (Use the Euler method, (1.7))
  - ▷  $t_{i+1} = t_i + \Delta t$ .
  - ▷ repeat for  $n - 1$  time steps

In Fortran this can be written as

#### Subroutine calculate in Fortran

---

```

      subroutine calculate(n_uranium,t,tau,dt,n)
c      Now use the Euler method
c      variable dimensioning is used for arrays n_uranium() and t()
      double precision n_uranium(n),t(n)
      do i = 1,n-1
          n_uranium(i+1) = n_uranium(i)-(n_uranium(i)/tau)*dt
          t(i+1) = t(i) + dt
      end do
      return
      end

```

---

This routine loops through each time step using the `do` and `continue` statements (other languages have analogous facilities to write simple loops). The key statement is the one in which `n_uranium(i+1)` is calculated. This statement contains *all* of the physics of the program and is closely analogous to (1.7). As mentioned above, the array `n_uranium` corresponds to the variable  $N_U(t)$ , and the value stored in the  $i$ th element of `n_uranium` is the number of uranium nuclei present at time  $t(i)$ .

The final subroutine `store` writes the result to a file. It uses a standard Fortran approach to store the results in the file `decay.dat`. The values of  $t$  and  $N_U$  are written as pairs, with a separate line for each value of  $t$ . The results can then be read from this file, in order to plot the results, or use them in a subsequent calculation.

#### Subroutine store in Fortran

---

```

      subroutine store(n_uranium,t,n)
      double precision n_uranium(n),t(n)
      open(1,file='decay.dat')
      do i=1,n
          write(1,20) t(i),n_uranium(i)
      end do
      close(1)
20  format(1x,1p,2(e12.5,2x))
      return
      end

```

---

For convenience, our complete Fortran program is listed in one piece below.

---

**Fortran version of radioactive decay program**

---

```

c Simulation of radioactive decay
c Program to accompany "Computational Physics" by N. Giordano/H. Nakanishi
      program decay
      declare the arrays we will need
      double precision n_uranium(100), t(100)
c      use subroutines to do the work
      call initialize(n_uranium,t,tau,dt,n)
      call calculate(n_uranium,t,tau,dt,n)
      call store(n_uranium,t,n)
      stop
      end
c
      subroutine initialize(nuclei,t,tc,dt,n)
c      Initialize variables
      double precision nuclei(1),t(1)
      print *, 'initial number of nuclei -> '
      read(5,*) nuclei(1)
      print *, 'time constant -> '
      read(5,*) tc
      print *, 'time step -> '
      read(5,*) dt
      print *, 'total time -> '
      read(5,*) time
      t(1) = 0
      n=min(int(time/dt),100)
      return
      end
c
      subroutine calculate(n_uranium,t,tau,dt,n)
c      Now use the Euler method
c      Variable dimensioning is used for arrays n_uranium() and t()
      double precision n_uranium(n),t(n)
      do i = 1,n-1
          n_uranium(i+1) = n_uranium(i)-(n_uranium(i)/tau)*dt
          t(i+1) = t(i) + dt
      end do
      return
      end
c
      subroutine store(n_uranium,t,n)
      double precision n_uranium(n),t(n)
      open(1,file='decay.dat')
      do i=1,n
          write(1,20) t(i),n_uranium(i)
      end do
      close(1)
      format(1x,1p,2(e12.5,2x))
      return
      end
20

```

---

We have spent a lot of time discussing a Fortran program for the radioactive decay problem, but the basic ideas — the basic structure of the program — can be implemented in many other languages. Below we give a program written in C that does the same calculation, using the same program structure with the same algorithm. It begins by declaring the necessary variables, then uses subroutines `initialize`, `calculate`, and `store`, as outlined in the pseudocode in Example 1.3. These subroutines are quite similar to those in the Fortran program given at left, and produce identical results.

---

**Radioactive decay program in C**

---

```

/* decay.c
 * Simulation of radioactive decay
 * Program to accompany "Computational Physics" by N. Giordano/H. Nakanishi
 */
#include <math.h>
#include <stdio.h>
#define MAX 100
double n_uranium[MAX]; /* number of uranium atoms */
double t[MAX];          /* store time values here */
double dt;              /* time step */
double tau;             /* decay time constant */
double t_max;           /* time to end simulation */

main()
{
    initialize(n_uranium,t,&tau,&dt);
    calculate(n_uranium,t,tau,dt);
    store(n_uranium,t);
}

/* initialize the variables */
initialize(nuclei,t,tc,dt)
double *nuclei,*t,*tc,*dt;
{
    printf("initial number of nuclei -> ");
    scanf("%lf",&(nuclei[0])); /* begin using arrays at index 0 */
    printf("time constant -> ");
    scanf("%lf",tc);
    printf("time step -> ");
    scanf("%lf",dt);
    t[0] = 0.0;
    return;
}

/* calculate the results and store them in the arrays t() and n_u() */
calculate(nuclei,t,tc,dt)
double *nuclei,*t,*tc,*dt;
{
    int i;
    for(i = 0; i < MAX-1; i++) {
        nuclei[i+1] = nuclei[i] - (nuclei[i] / tc) * dt;
        t[i+1] = t[i] + dt;
    }
}

```

```

    }
    return;
}
/* save the results to a file */
store(nuclei,t)
double *nuclei,*t;
{
    FILE *fp_out;
    int i;
    fp_out = fopen("decay.dat","w");
    for(i = 0; i < MAX; i++) {
        fprintf(fp_out,"%g\t%g\n",t[i],nuclei[i]);
    }
    fclose(fp_out);
    return;
}

```

We have given specific example programs in Fortran and C, as these are languages that we (the authors) use in our everyday work.<sup>7</sup> However, this certainly does *not* mean that one of these languages is the best choice for you; that is a judgment that you must make.

Next we come to an *extremely* important point, that is a key issue in nearly all the problems we will discuss in this book. Our program `decay` calculates how  $N_U$  varies with time, and puts the results as numbers into a file. However, we still have the job of *understanding* the results. In this problem, and in most cases, this job of understanding is best done by examining the results in graphical form. While there will be times when the result of a calculation can be expressed or conveyed as one or two numbers, it will usually happen that our calculations produce a *lot* of numbers. Making sense of such results is almost always simplest when the results are displayed graphically. In our decay problem we will want to make a graph of  $N_U$  as a function of  $t$ . Methods for producing a graphical display, either on a video display or on paper, can vary *a lot* from one computer platform and language to another, and there is no way that we can possibly give a full discussion of how you should go about it on your particular system. Some languages have very powerful graphics capabilities “built-in” (e.g., Matlab and Mathematica), while in other cases you may need to use separate graphics programs to display the results in the file `decay.dat`.

The ability to easily display results in graphical form is *absolutely essential* for work in computational physics. Fortunately, virtually all computer systems have the programs you will need (so you will not have to write your own!). Before you go much further in this book we urge you to learn how to use the graphics capabilities of your particular system.

Figure 1.1 shows an example of the output produced by our radioactive decay program, along with the exact result, Equation (1.2). Here we have used the initial values  $n_{\text{uranium}}(1) = 100$  and  $t(1) = 0$ , along with a time constant of 1 s and a time step of 0.05 s. We will consider the choice of time step later. For now we note only that our calculated values compare well with the exact result.

<sup>7</sup>This may also give you some idea of when we first learned to program.

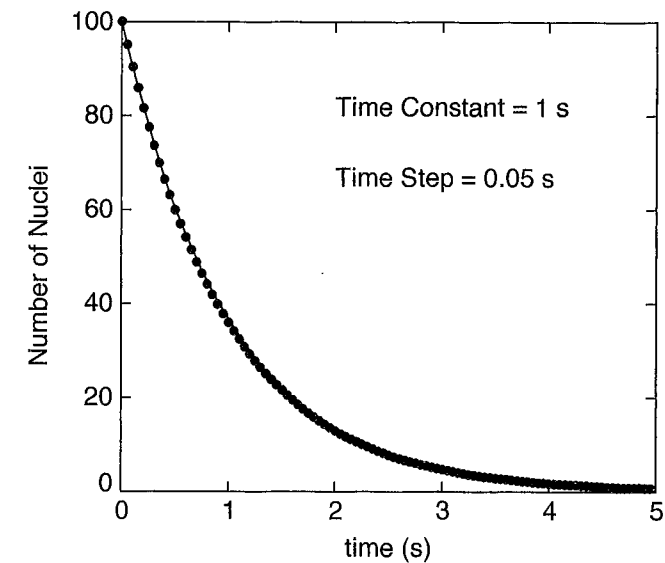


FIGURE 1.1: Circles indicate the numerical solution to the radioactive decay problem. The solid line is the exact solution Equation (1.2).

## 1.4 TESTING YOUR PROGRAM

The title of Section 1.3 was a little misleading. It was concerned with producing a program that runs without any errors in the syntax, etc., such as undeclared arrays, spelling errors, or variables omitted when calling a subroutine. However, we should not really consider it to be a working program until we are convinced that its output is correct! Checking a program is not always a trivial task, but there are some general guidelines. After a program has been debugged so that it can be run without any complaints about the syntax, there are several things you should do to verify that the results of the program are correct.

*Does the output look reasonable?* Before you perform any calculation you should always have at least a rough idea of what the result should be. The first thing you should do when considering the results from any program is ask whether or not they are consistent with your intuition and instincts. This exercise can also improve your overall understanding of the problem. When you show your result to someone else, you should always be able to convince them that it makes sense.<sup>8</sup>

*Does your program agree with any exact results that are available?* Since we knew the analytic solution for our radioactive decay problem, we were able to compare our numerical values with the exact result. While such a comparison will not be possible for most of the numerical calculations you will encounter, exact results are

<sup>8</sup>One definition of a physicist is “a person who can calculate anything to within an order of magnitude.” This is perhaps a bit overstated, but you should usually be able to anticipate an answer with this sort of accuracy.

sometimes available in certain limits, that is, for special values of the parameters. You should always run your program in those limits to check that it gives the correct answer. This is a necessary (but not sufficient) test that a program is correct in the general case.

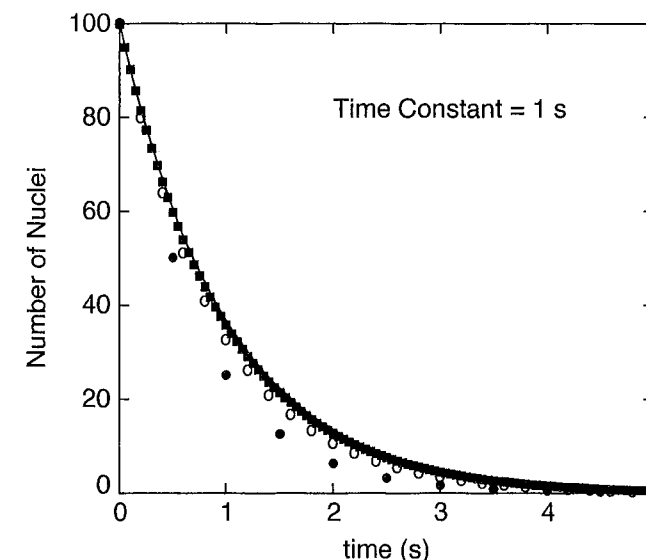
*Always check that your program gives the same answer for different “step sizes.”* Our decay program involved a time-step variable,  $\Delta t$ , and most other numerical calculations involve similar step- or grid-size parameters. Your final answer should be independent of the values of such parameters. This is another necessary (but not sufficient) test of a program’s accuracy.

Checking your program should not be viewed as a trivial, last-minute job. It is not unreasonable to spend as much time checking (and correcting) a program as it takes writing it. After all, a result is not much good if you don’t trust it to be correct.

## 1.5 NUMERICAL CONSIDERATIONS

The issue of numerical errors is central to the computational solution of any problem. Indeed, this is such an important topic that there are *many* books devoted to this area. Questions such as how to design or choose the best algorithm for a particular problem, and how to estimate the numerical errors associated with an algorithm, are central topics in many computer science and applied mathematics courses. We could easily spend *a lot* of space discussing the Euler method and all the other algorithms we will be using in this book. However, while this would certainly be an instructive thing to do, it would leave us with very little time to explore the *physics* that can be done with these algorithms. In this book our emphasis will be on the physics, rather than the numerical methods. This does not mean that we view numerical methods as unimportant, but just that there is only so much that we can do in one book. Nevertheless, we will be making some comments about the choice of algorithms, their numerical uncertainties, and their stability. We will attempt to blend these discussions with the physics as we go, particularly when they are closely related to the physics of the problem. The Appendices contain more detailed and systematic discussions of many of the numerical methods we use in this book.

With our radioactive decay program, errors were introduced by the *approximation* used to estimate the solution of the differential equation (1.7). Errors are also produced by the finite numerical precision in any programming language, including Fortran and C. These are known as round-off errors, and are almost always present when numbers are represented by a finite number of digits. Fortunately, this is not usually a severe problem, as most modern computer language systems employ a large number of significant digits. In order to be as safe as possible you should always use double precision variables in Fortran, C, and other languages where this is an option. However, this is no guarantee that these errors will be negligible, as the numerical solutions to certain types of problems are inherently sensitive to round-off errors. It is hard to give any general rules about what to watch for, but we will discuss such difficulties when they arise in several problems later in this book.



**FIGURE 1.2:** Numerical solution of the radioactive decay problem using the Euler method for different values of the time step. The time constant was 1 s, and the time steps were: filled circles, 0.5 s; open circles, 0.2 s; squares, 0.05 s. The solid curve is the exact solution.

In attacking the radioactive decay problem we were led to treat time as a discrete variable; that is, we converted the differential equation (1.1) into a difference equation (1.7), which enabled us to compute estimates for  $N_U$  at the discrete times  $n\Delta t$ , where  $n$  is an integer. Such a “discretization” of time, or space, or both, is a common practice in numerical calculations, and brings up two related questions: (1) How do we know that the errors introduced by this discreteness are negligible, and (2) How do we choose the value of such a step size for a calculation? Again, there are no general answers to these questions. All we can do is give some guidelines, which can be illustrated using our radioactive decay problem.

The first guideline is that a calculation should *always* be repeated using several different values of the step size. Figure 1.2 shows the results from our radioactive decay calculation using three different values of the time step. The time constant  $\tau$  was taken to be 1 s, and the symbols show the numerical results for time steps of 0.5, 0.2, and 0.05 s. Also shown is the exact solution (the solid line). Because of the nonzero step size, our program only yields results at discrete values of  $t$ , which is why the calculated points become more dense as the time step is reduced. We see that as the time step is made smaller, the calculated values converge quickly to the exact solution. This is not surprising, since intuition tells us that our approximations should become better as  $\Delta t$  is reduced. Being more quantitative, we already noted that the terms dropped in our derivation of (1.6) are of order  $(\Delta t)^2$ . In order to understand the accuracy of the calculated result at a specific value of  $t$ , we note that it takes  $t/\Delta t$  time steps to reach this point (starting from  $t = 0$ ). The total

error (often termed the *global error*) is proportional to the product of the number of time steps ( $t/\Delta t$ ) and the error per step ( $\sim (\Delta t)^2$ ), and is thus of order  $\Delta t$ . Hence, we expect the difference between the numerical (Euler) and exact results to be cut in half for each factor of 2 reduction of  $\Delta t$ . It is clear from Figure 1.2 that the numerical results become extremely close (on the scale of this figure) to the exact solution for time steps below about 0.05 s.

For our decay problem we have the exact solution, which makes it easy to evaluate the accuracy of our numerical results. However, we will not usually be this fortunate, so it is important to consider how to evaluate our results when the exact result is not available. In such cases you should always check that the calculated result converges to a fixed value (or curve) as the step size is made smaller. A closely related matter concerns the choice of step size in the first place; that is, what is a reasonable value? Here, yet again, there are no hard and fast rules. Ideally, you should use a step size that is small compared to any characteristic time scales in the problem. In our decay problem this time scale is  $\tau$ , since its value determines how fast  $N_U$  varies. Our recommendation is that you pick the time step to be a small fraction of  $\tau$ . We see from Figure 1.2 that a time step that is a few percent (or less) of this characteristic scale is a good choice.

One more important issue concerning numerical precision will be encountered later in this book. There are situations in which what appears to be a very reasonable numerical approach can be inherently unstable. For example, we will find in Chapter 3 that the Euler method, which we have found to work well for the radioactive decay problem, fails miserably when applied to problems involving oscillatory motion. The message here is that there is no one *best* method for solving ordinary differential equations or any other particular class of problems. You must apply your understanding of the theory behind an algorithm (as discussed, for example, in a numerical methods book such as Press et al. [1986]) and your ingenuity (and also this book!) to choose the appropriate method for any particular problem.

## 1.6 PROGRAMMING GUIDELINES AND PHILOSOPHY

In our construction of the radioactive decay program we noted that writing a program is necessarily an individualistic endeavor. Nevertheless, there are certain guidelines that are generally recommended, some of which were touched on above. All of these guidelines have essentially the same purpose: to make the program as easy to understand as possible.

- *Program structure.* Use subroutines to organize the major tasks and make the program more readable and understandable. The main program for the decay problem was basically an outline of the program; we recommend this style for *all* programs. Use subroutines and functions to perform any jobs that take more than a few lines of code, or that are required repeatedly.
- *Use descriptive names.* Choose the names of variables and subroutines according to the problem at hand. Descriptive names make a program easier to understand, as they act as built-in comment statements.

- *Use comment statements.* Include comment statements to explain program logic and describe variables. A short subroutine that uses descriptive variable names should not need a large number of comment statements.
- *Sacrifice (almost) everything for clarity.* It is often tempting to write a critical piece of code in a very compact or terse manner in the misguided belief that this will make the program run faster. While a compact code may occasionally run faster, this is not a reliable indication of computational efficiency. Moreover, such compact code will nearly always have a cost in terms of clarity and readability; there are many famous pieces of code in Fortran and C that come to mind here. It is almost always better to take a few more lines, or a few more variables, to do a job, if it makes the code more understandable.<sup>9</sup> It is also a good idea to make the code look as much as possible like the associated physics equation. This will save time later when you are checking your program.

The execution speed becomes an important programming issue only when the overall computer time requirements become substantial (say, over several minutes). Even then, significant savings can usually be made only by algorithmic improvements and it is rare to see a noticeable gain in efficiency due to fiddling with a line of code here or there. In this book we will generally write our programs without worrying much about program efficiency. If you want to “tuneup” a program to make it run faster, this should be done only *after* it is known to be correct. Moreover, modern language compilers often do a good job of optimizing code without the programmer even knowing it!

- *Take time to make graphical output as clear as possible.* Think carefully about what quantities to plot and in what manner.<sup>10</sup> The axes should be labeled clearly (including units, where appropriate), and parameter values given directly on the graph.

## EXERCISES

*Note:* Throughout this book, exercises labeled with an \* are more challenging than others in that section.

- 1.1. The velocity of a freely falling object near Earth’s surface is described by the equation

$$\frac{dv}{dt} = -g, \quad (1.8)$$

where  $v$  is the velocity and  $g = 9.8 \text{ m/s}^2$  is the acceleration due to gravity. Write a program that employs the Euler method to compute the solution to (1.8); that is, calculate  $v$  as a function of  $t$ . For simplicity, assume that the initial velocity is zero—that is, the object starts from rest—and calculate the solution for times  $t = 0$  to  $t = 10$  s. Repeat the calculation for several different values of the time step, and compare the results with the exact solution to (1.8). It turns out

<sup>9</sup>In very large-scale projects, there may perhaps come a time when *clarity* begins to compete with *computational efficiency*. If so, some compromises may need to be made; however, in all cases, *clarity* should receive an extremely high priority.

<sup>10</sup>Sometimes, plotting either or both axes logarithmically may be more appropriate than using linear scales, see Chapter 3 for some examples.



that for this case the Euler method gives the exact result. Verify this with your numerical results and prove it analytically.

- 1.2. The position of an object moving horizontally with a constant velocity,  $v$ , is described by the equation

$$\frac{dx}{dt} = v. \quad (1.9)$$

Assuming that the velocity is a constant, say  $v = 40$  m/s, use the Euler method to solve (1.9) for  $x$  as a function of time. Compare your result with the exact solution.

- 1.3. It is often the case that the frictional force on an object will increase as the object moves faster. A fortunate example of this is a parachutist; the role of the parachute is to produce a frictional force due to air drag, which is larger than would normally be the case without the parachute. The physics of air drag will be discussed in more detail in the next chapter. Here we consider a very simple example in which the frictional force depends on the velocity. Assume that the velocity of an object obeys an equation of the form

$$\frac{dv}{dt} = a - bv, \quad (1.10)$$

where  $a$  and  $b$  are constants. You could think of  $a$  as coming from an applied force, such as gravity, while  $b$  arises from friction. Note that the frictional force is negative (we assume that  $b > 0$ ), so that it opposes the motion, and that it increases in magnitude as the velocity increases. Use the Euler method to solve (1.10) for  $v$  as a function of time. A convenient choice of parameters is  $a = 10$  and  $b = 1$ . You should find that  $v$  approaches a constant value at long times; this is called the terminal velocity.

- \*1.4. Consider a radioactive decay problem involving two types of nuclei,  $A$  and  $B$ , with populations  $N_A(t)$  and  $N_B(t)$ . Suppose that type  $A$  nuclei decay to form type  $B$  nuclei, which then also decay, according to the differential equations

$$\begin{aligned} \frac{dN_A}{dt} &= -\frac{N_A}{\tau_A}, \\ \frac{dN_B}{dt} &= \frac{N_A}{\tau_A} - \frac{N_B}{\tau_B}, \end{aligned} \quad (1.11)$$

where  $\tau_A$  and  $\tau_B$  are the decay time constants for each type of nucleus. Use the Euler method to solve these coupled equations for  $N_A$  and  $N_B$  as functions of time. This problem can also be solved exactly, as was the case with our original nuclear decay problem (1.1). Obtain the analytic solutions for  $N_A(t)$  and  $N_B(t)$ , and compare them with your numerical results. It is also interesting to explore the behavior found for different values of the ratio  $\tau_A/\tau_B$ . In particular, try to interpret the short and long time behaviors for different values of this ratio.

- \*1.5. Consider again a decay problem with two types of nuclei  $A$  and  $B$ , but now suppose that nuclei of type  $A$  decay into ones of type  $B$ , while nuclei of type  $B$  decay into ones of type  $A$ . Strictly speaking, this is not a “decay” process, since it is possible for the type  $B$  nuclei to turn back into type  $A$  nuclei. A better analogy would be a resonance in which a system can tunnel or move back and forth between two states  $A$  and  $B$  which have equal energies. The corresponding

rate equations are

$$\begin{aligned} \frac{dN_A}{dt} &= \frac{N_B}{\tau} - \frac{N_A}{\tau}, \\ \frac{dN_B}{dt} &= \frac{N_A}{\tau} - \frac{N_B}{\tau}, \end{aligned} \quad (1.12)$$

where for simplicity we have assumed that the two types of decay are characterized by the same time constant,  $\tau$ . Solve this system of equations for the numbers of nuclei,  $N_A$  and  $N_B$ , as functions of time. Consider different initial conditions, such as  $N_A = 100$ ,  $N_B = 0$ , etc., and take  $\tau = 1$  s. Show that your numerical results are consistent with the idea that the system reaches a steady state in which  $N_A$  and  $N_B$  are constant. In such a steady state, the time derivatives  $dN_A/dt$  and  $dN_B/dt$  should vanish.

- 1.6. Population growth problems often give rise to rate equations that are first-order. For example, the equation

$$\frac{dN}{dt} = aN - bN^2, \quad (1.13)$$

might describe how the number of individuals in a population,  $N$ , varies with time. Here the first term  $aN$  corresponds to the birth of new members, while the second term  $-bN^2$  corresponds to deaths. The death term is proportional to  $N^2$  to allow for the fact that food will become harder to find when the population  $N$  becomes large. Begin by solving (1.13) with  $b = 0$  using the Euler method, and compare your numerical result with the exact solution. Then solve (1.13) with nonzero values  $b$ . Give an intuitive explanation of your results. Interesting values of  $a$  and  $b$  depend on the initial population  $N$ . For small  $N(0)$ ,  $a = 10$  and  $b = 3$  is a good choice, while for  $N(0) = 1000$  a good choice is  $a = 10$  and  $b = 0.01$ .

## REFERENCES

- [1] B. W. Kernighan and P. J. Plauger, 1978, *The Elements of Programming Style*, 2d ed., McGraw-Hill, New York. A very nice discussion of how to write clear, readable, and efficient programs in any language.
- [2] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, 1986, *Numerical Recipes*, Cambridge University Press, Cambridge. An excellent all-purpose reference on numerical methods and why they work.