CHAPTER 4

ACCURACY AND SPEED

WE HAVE now seen the basic elements of programming in Python: input and output, variables and arithmetic, loops and if statements. With these we can perform a wide variety of calculations. We have also seen how to visualize our results using various types of computer graphics. There are many additional features of the Python language that we haven't covered. In later chapters of the book, for example, we will introduce a number of specialized features, such as facilities for doing linear algebra and Fourier transforms. But for now we have the main components in place to start doing physics.

There is, however, one fundamental issue that we have not touched upon. Computers have limitations. They cannot store real numbers with an infinite number of decimal places. There is a limit to the largest and smallest numbers they can store. They can perform calculations quickly, but not infinitely quickly. In many cases these issues need not bother us—the computer is fast enough and accurate enough for many of the calculations we do in physics. However, there are also situations in which the computer's limitations will affect us significantly, so it will be crucial that we understand those limitations, as well as methods for mitigating them or working around them when necessary.

4.1 VARIABLES AND RANGES

We have seen examples of the use of variables in computer programs, including integer, floating-point, and complex variables, as well as lists and arrays. Python variables can hold numbers that span a wide range of values, including very large numbers, but they cannot hold numbers that are arbitrarily large. For instance, the largest value you can give a floating-point variable is about 10^{308}. (There is also a corresponding largest negative value of about -10^{308}.) This is enough for most physics calculations, but we will see occasional ex-

amples where we run into problems. Complex numbers are similar: both their real and imaginary parts can go up to about $\pm 10^{308}$ but not larger.[1] Large numbers can be specified in scientific notation, using an "`e`" to denote the exponent. For instance, `2e9` means 2×10^9 and `1.602e-19` means 1.602×10^{-19}. Note that numbers specified in scientific notation are always floats. Even if the number is, mathematically speaking, an integer (like `2e9`), the computer will still treat it as a float.

If the value of a variable exceeds the largest floating-point number that can be stored on the computer we say the variable has *overflowed*. For instance, if a floating-point variable `x` holds a number close to the maximum allowed value of 10^{308} and then we execute a statement like "`y = 10*x`" it is likely that the result will be larger than the maximum and the variable `y` will overflow (but not the variable `x`, whose value is unchanged).

If this happened in the course of a calculation you might imagine that the program would stop, perhaps giving an error message, but in Python this is not what happens. Instead the computer will set the variable to the special value "`inf`," which means infinity. If you print such a variable with a print statement, the computer will actually print the word "`inf`" on the screen. In effect, every number over 10^{308} is infinity as far as the computer is concerned. Unfortunately, this is usually not what you want, and when it happens your program will probably give incorrect answers, so you need to watch out for this problem. It's rare, but it'll probably happen to you at some point.

There is also a smallest number (meaning smallest magnitude) that can be represented by a floating-point variable. In Python this number is 10^{-308} roughly.[2] If you go any smaller than this—if the calculation *underflows*—the computer will just set the number to zero. Again, this usually messes things up and gives wrong answers, so you need to be on the lookout.

What about integers? Here Python does something clever. There is no largest integer value in Python: it can represent integers to *arbitrary precision*. This means that no matter how many digits an integer has, Python stores all of them—provided you have enough memory on your computer. Be aware, however, that calculations with integers, even simple arithmetic operations, take longer with more digits, and can take a very long time if there are very

[1] The actual largest number is 1.79769×10^{308}, which is the decimal representation of the binary number 2^{1024}, the largest number that can be represented in the IEEE 754 double-precision floating-point format used by the Python language.

[2] Actually 2.22507×10^{-308}, which is 2^{-1022}.

many digits. Try, for example, doing `print(2**1000000)` in Python. The calculation can be done—it yields a number with 301 030 digits—but it's so slow that you might as well forget about using your computer for anything else for the next few minutes.[3]

Exercise 4.1: Write a program to calculate and print the factorial of a number entered by the user. If you wish you can base your program on the user-defined function for factorial given in Section 2.6, but write your program so that it calculates the factorial using *integer* variables, not floating-point ones. Use your program to calculate the factorial of 200.

Now modify your program to use floating-point variables instead and again calculate the factorial of 200. What do you find? Explain.

4.2 NUMERICAL ERROR

Floating-point numbers (unlike integers) are represented on the computer to only a certain precision. In Python, at least at the time of writing of this book, the standard level of precision is 16 significant digits. This means that numbers like π or $\sqrt{2}$, which have an infinite number of digits after the decimal point, can only be represented approximately. Thus, for instance:

True value of π:	3.1415926535897932384626 ...
Value in Python:	3.141592653589793
Difference:	0.0000000000000002384626 ...

The difference between the true value of a number and its value on the computer is called the *rounding error* on the number. It is the amount by which the computer's representation of the number is wrong.

A number does not have to be irrational like π to suffer from rounding error—any number whose true value has more than 16 significant figures will get rounded off. What's more, when one performs arithmetic with floating-point numbers, the answers are only guaranteed accurate to about 16 figures, even if the numbers that went into the calculation were expressed exactly. If

[3]If you do actually try this, then you might want to know how to stop your program if you get bored waiting for it to finish. The simplest thing to do is just to close the window where it's running.

you add 1.1 and 2.2 in Python, then obviously the answer should be 3.3, but the computer might give 3.2999999999999999 instead.

Usually this is accurate enough, but there are times when it can cause problems. One important consequence of rounding error is that you should *never use an if statement to test the equality of two floats.* For instance, you should never, in any program, have a statement like

```
if x==3.3:
    print(x)
```

because it may well not do what you want it to do. If the value of `x` is supposed to be 3.3 but it's actually 3.2999999999999999, then as far as the computer is concerned it's not 3.3 and the if statement will fail. In fact, it rarely occurs in physics calculations that you need to test the equality of floats, but if you do, then you should do something like this instead:

```
epsilon = 1e-12
if abs(x-3.3)<epsilon:
    print(x)
```

As we saw in Section 2.2.6, the built-in function `abs` calculates the absolute value of its argument, so `abs(x-3.3)` is the absolute difference $|x - 3.3|$. The code above tests whether this difference is less than the small number `epsilon`. In other words, the if statement will succeed whenever `x` is very close to 3.3, but the two don't have to be exactly equal. If `x` is 3.2999999999999999 things will still work as expected. The value of `epsilon` has to be chosen appropriately for the situation—there's nothing special or universal about the value of 10^{-12} used above and a different value may be appropriate in another calculation.

The rounding error on a number, which we will denote ϵ, is defined to be the amount you would have to add to the value calculated by the computer to get the true value. For instance, if we do the following:

```
from math import sqrt
x = sqrt(2)
```

then we will end up not with $x = \sqrt{2}$, but rather with $x + \epsilon = \sqrt{2}$, where ϵ is the rounding error, or equivalently $x = \sqrt{2} - \epsilon$. This is the same definition of error that one uses when discussing measurement error in experiments. When we say, for instance, that the age of the universe is 13.80 ± 0.04 billion years, we mean that the measured value is 13.80 billion years, but the true value is possibly greater or less than this by an amount of order 0.04 billion years.

The error ϵ in the example above could be either positive or negative, depending on how the variable x gets rounded off. If we are lucky ϵ could be small, but we cannot count on it. In general if x is accurate to a certain number of significant digits, say 16, then the rounding error will have a typical size of $x/10^{16}$. It's usually a good assumption to consider the error to be a (uniformly distributed) random number with standard deviation $\sigma = Cx$, where $C \simeq 10^{-16}$ in this case. We will refer to the constant C as the *error constant*. When quoting the error on a calculation we typically give the value of the standard deviation σ. (We can't give the value of the error ϵ itself, since we don't know it—if we did, then we could calculate $x + \epsilon$ and recover the exact value for the quantity of interest, so there would in effect be no error in the calculation at all.)

Rounding error is important, as described above, if we are testing the equality of two floating-point numbers, but in other respects it may appear to be only a minor annoyance. An error of one part in 10^{16} does not seem very bad. But what happens if we now add, subtract, or otherwise combine several different numbers, each with its own error? In many ways the rounding error on a number behaves similarly to measurement error in a laboratory experiment, and the rules for combining errors are the same. For instance, if we add or subtract two numbers x_1 and x_2, with standard deviations σ_1 and σ_2, then standard results about combinations of random variables tell us that the *variance* σ^2 of the sum or difference is equal to the sum of the individual variances:

$$\sigma^2 = \sigma_1^2 + \sigma_2^2. \tag{4.1}$$

Hence the standard deviation of the sum or difference is

$$\sigma = \sqrt{\sigma_1^2 + \sigma_2^2}. \tag{4.2}$$

Similarly if we multiply or divide two numbers then the variance of the result x obeys

$$\frac{\sigma^2}{x^2} = \frac{\sigma_1^2}{x_1^2} + \frac{\sigma_2^2}{x_2^2}. \tag{4.3}$$

But, as discussed above, the standard deviations on x_1 and x_2 are given by $\sigma_1 = Cx_1$ and $\sigma_2 = Cx_2$, so that if, for example, we are adding or subtracting our two numbers, meaning Eq. (4.2) applies, then

$$\sigma = \sqrt{C^2x_1^2 + C^2x_2^2} = C\sqrt{x_1^2 + x_2^2}. \tag{4.4}$$

I leave it as an exercise to show that the corresponding result for the error on the product of two numbers $x = x_1 x_2$ or their ratio $x = x_1/x_2$ is

$$\sigma = \sqrt{2}\, Cx. \tag{4.5}$$

We can extend these results to combinations of more than two numbers. If, for instance, we are calculating the sum of N numbers $x_1 \ldots x_N$ with errors having standard deviation $\sigma_i = Cx_i$, then the variance on the final result is the sum of the variances on the individual numbers:

$$\sigma^2 = \sum_{i=1}^{N} \sigma_i^2 = \sum_{i=1}^{N} C^2 x_i^2 = C^2 N \overline{x^2}, \tag{4.6}$$

where $\overline{x^2}$ is the mean-square value of x. Thus the standard deviation on the final result is

$$\sigma = C\sqrt{N}\,\sqrt{\overline{x^2}}. \tag{4.7}$$

As we can see, this quantity increases in size as N increases—the more numbers we combine, the larger the error on the result—although the increase is a relatively slow one, proportional to the square root of N.

We can also ask about the *fractional* error on $\sum_i x_i$, i.e., the total error divided by the value of the sum. The size of the fractional error is given by

$$\frac{\sigma}{\sum_i x_i} = \frac{C\sqrt{N\,\overline{x^2}}}{N\,\overline{x}} = \frac{C}{\sqrt{N}}\,\frac{\sqrt{\overline{x^2}}}{\overline{x}}, \tag{4.8}$$

where $\overline{x} = N^{-1} \sum_i x_i$ is the mean value of x. In other words the fractional error in the sum actually *goes down* as we add more numbers.

At first glance this appears to be pretty good. So what's the problem? Actually, there are a couple of them. One is when the sizes of the numbers you are adding vary widely. If some are much smaller than others then the smaller ones may get lost. But the most severe problems arise when you are not adding but subtracting numbers. Suppose, for instance, that we have the following two numbers:

$$x = 1000000000000000$$
$$y = 1000000000000001.2345678901234$$

and we want to calculate the difference $y - x$. Unfortunately, the computer only represents these two numbers to 16 significant figures, which means that

as far as the computer is concerned:

$$x = 100000000000000$$
$$y = 100000000000001.2$$

The first number is represented exactly in this case, but the second has been truncated. Now when we take the difference we get $y - x = 1.2$, when the true result would be 1.2345678901234. In other words, instead of 16-figure accuracy, we now only have two figures and the fractional error is several percent of the true value. This is much worse than before.

To put this in more general terms, if the difference between two numbers is very small, comparable with the error on the numbers, i.e., with the accuracy of the computer, then the fractional error can become large and you may have a problem.

EXAMPLE 4.1: THE DIFFERENCE OF TWO NUMBERS

To see an example of this in practice, consider the two numbers

$$x = 1, \qquad y = 1 + 10^{-14}\sqrt{2}. \tag{4.9}$$

Trivially we see that

$$10^{14}(y - x) = \sqrt{2}. \tag{4.10}$$

Let us perform the same calculation in Python and see what we get. Here is the program:

```
from math import sqrt
x = 1.0
y = 1.0 + (1e-14)*sqrt(2)
print((1e14)*(y-x))
print(sqrt(2))
```

The penultimate line calculates the value in Eq. (4.10) while the last line prints out the true value of $\sqrt{2}$ (at least to the accuracy of the computer). Here's what we get when we run the program:

```
1.42108547152
1.41421356237
```

As we can see, the calculation is accurate to only the first decimal place—after that the rest is garbage.

This issue, of large errors in calculations that involve the subtraction of numbers that are nearly equal, arises with some frequency in physics calculations. We will see various examples throughout the book. It is perhaps the most common cause of significant numerical error in computations and you need to be aware of it at all times when writing programs.

Exercise 4.2: Quadratic equations

Consider a quadratic equation $ax^2 + bx + c = 0$ that has real solutions.

a) Write a program that takes as input the three numbers, a, b, and c, and prints out the two solutions using the standard formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Use your program to compute the solutions of $0.001x^2 + 1000x + 0.001 = 0$.

b) There is another way to write the solutions to a quadratic equation. Multiplying top and bottom of the solution above by $-b \mp \sqrt{b^2 - 4ac}$, show that the solutions can also be written as

$$x = \frac{2c}{-b \mp \sqrt{b^2 - 4ac}}.$$

Add further lines to your program to print these values in addition to the earlier ones and again use the program to solve $0.001x^2 + 1000x + 0.001 = 0$. What do you see? How do you explain it?

c) Using what you have learned, write a new program that calculates both roots of a quadratic equation accurately in all cases.

This is a good example of how computers don't always work the way you expect them to. If you simply apply the standard formula for the quadratic equation, the computer will sometimes get the wrong answer. In practice the method you have worked out here is the correct way to solve a quadratic equation on a computer, even though it's more complicated than the standard formula. If you were writing a program that involved solving many quadratic equations this method might be a good candidate for a user-defined function: you could put the details of the solution method inside a function to save yourself the trouble of going through it step by step every time you have a new equation to solve.

Exercise 4.3: Calculating derivatives

Suppose we have a function $f(x)$ and we want to calculate its derivative at a point x. We can do that with pencil and paper if we know the mathematical form of the function, or we can do it on the computer by making use of the definition of the derivative:

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \lim_{\delta \to 0} \frac{f(x+\delta) - f(x)}{\delta}.$$

On the computer we can't actually take the limit as δ goes to zero, but we can get a reasonable approximation just by making δ small.

a) Write a program that defines a function `f(x)` returning the value $x(x-1)$, then calculates the derivative of the function at the point $x = 1$ using the formula above with $\delta = 10^{-2}$. Calculate the true value of the same derivative analytically and compare with the answer your program gives. The two will not agree perfectly. Why not?
b) Repeat the calculation for $\delta = 10^{-4}$, 10^{-6}, 10^{-8}, 10^{-10}, 10^{-12}, and 10^{-14}. You should see that the accuracy of the calculation initially gets better as δ gets smaller, but then gets worse again. Why is this?

We will look at numerical derivatives in more detail in Section 5.10, where we will study techniques for dealing with these issues and maximizing the accuracy of our calculations.

4.3 PROGRAM SPEED

As we have seen, computers are not infinitely accurate. And neither are they infinitely fast. Yes, they work at amazing speeds, but many physics calculations require the computer to perform millions of individual computations to get a desired overall result and collectively those computations can take a significant amount of time. Some of the example calculations described in Chapter 1 took months to complete, even though they were run on some of the most powerful computers in the world.

One thing we need to get a feel for is how fast computers really are. As a general guide, performing a million mathematical operations is no big problem for a computer—it usually takes less than a second. Adding a million numbers together, for instance, or finding a million square roots, can be done in very little time. Performing a billion operations, on the other hand, could take minutes or hours, though it's still possible provided you are patient. Performing a trillion operations, however, will basically take forever. So a fair rule of thumb is that the calculations we can perform on a computer are ones that can be done with *about a billion operations or less.*

This is only a rough guide. Not all operations are equal and it makes a difference whether we are talking about additions or multiplications of single numbers (which are easy and quick) versus, say, calculating Bessel functions or multiplying matrices (which are not). Moreover, the billion-operation rule will change over time because computers get faster. However, computers have been getting faster a lot less quickly in the last few years—progress has slowed. So we're probably stuck with a billion operations for a while.

EXAMPLE 4.2: QUANTUM HARMONIC OSCILLATOR AT FINITE TEMPERATURE

The quantum simple harmonic oscillator has energy levels $E_n = \hbar\omega(n + \frac{1}{2})$, where $n = 0, 1, 2, \ldots, \infty$. As shown by Boltzmann and Gibbs, the average energy of a simple harmonic oscillator at temperature T is

$$\langle E \rangle = \frac{1}{Z} \sum_{n=0}^{\infty} E_n \mathrm{e}^{-\beta E_n}, \tag{4.11}$$

where $\beta = 1/(k_B T)$ with k_B being the Boltzmann constant, and $Z = \sum_{n=0}^{\infty} \mathrm{e}^{-\beta E_n}$. Suppose we want to calculate, approximately, the value of $\langle E \rangle$ when $k_B T = 100$. Since the terms in the sums for $\langle E \rangle$ and Z dwindle in size quite quickly as n becomes large, we can get a reasonable approximation by taking just the first 1000 terms in each sum. Working in units where $\hbar = \omega = 1$, here's a program to do the calculation:

File: `qsho.py`

```
from math import exp

terms = 1000
beta = 1/100
S = 0.0
Z = 0.0
for n in range(terms):
    E = n + 0.5
    weight = exp(-beta*E)
    S += weight*E
    Z += weight

print(S/Z)
```

Note a few features of this program:

1. Constants like the number of terms and the value of β are assigned to variables at the beginning of the program. As discussed in Section 2.7, this is good programming style because it makes them easy to find and modify and makes the rest of the program more readable.
2. We used just one for loop to calculate both sums. This saves time, making the program run faster.
3. Although the exponential $\mathrm{e}^{-\beta E_n}$ occurs separately in both sums, we calculate it only once each time around the loop and save its value in the variable `weight`. This also saves time: exponentials take significantly longer to calculate than, for example, additions or multiplications. (Of course

"longer" is relative—the times involved are probably still less than a microsecond. But if one has to go many times around the loop even those short times can add up.)

If we run the program we get this result:

```
99.9554313409
```

The calculation (on my desktop computer) takes 0.01 seconds. Now let us try increasing the number of terms in the sums (which just means increasing the value of the variable `terms` at the top of the program). This will make our approximation more accurate and give us a better estimate of our answer, at the expense of taking more time to complete the calculation. If we increase the number of terms to a million then it does change our answer somewhat:

```
100.000833332
```

The calculation now takes 1.4 seconds, which is significantly longer, but still a short time in absolute terms.

Now let's increase the number of terms to a billion. When we do this the calculation takes 22 minutes to finish, but the result does not change at all:

```
100.000833332
```

There are three morals to this story. First, a billion operations is indeed doable—if a calculation is important to us we can probably wait twenty minutes for an answer. But it's approaching the limit of what is reasonable. If we increased the number of terms in our sum by another factor of ten the calculation would take 220 minutes, or nearly four hours. A factor of ten beyond that and we'd be waiting a couple of days for an answer.

Second, there is a balance to be struck between time spent and accuracy. In this case it was probably worthwhile to calculate a million terms of the sum—it didn't take long and the result was noticeably, though not wildly, different from the result for a thousand terms. But the change to a billion terms was clearly not worth the effort—the calculation took much longer to complete but the answer was exactly the same as before. We will see plenty of further examples in this book of calculations where we need to find an appropriate balance between speed and accuracy.

Third, it's pretty easy to write a program that will take forever to finish. If we set the program above to calculate a trillion terms, it would take weeks to run. So it's worth taking a moment, before you spend a whole lot of time

writing and running a program, to do a quick estimate of how long you expect your calculation to take. If it's going to take a year then it's not worth it: you need to find a faster way to do the calculation, or settle for a quicker but less accurate answer. The simplest way to estimate running time is to make a rough count of the number of mathematical operations the calculation will involve; if the number is significantly greater than a billion, you have a problem.

EXAMPLE 4.3: MATRIX MULTIPLICATION

Suppose we have two $N \times N$ matrices represented as arrays `A` and `B` on the computer and we want to multiply them together to calculate their matrix product. Here is a fragment of code to do the multiplication and place the result in a new array called `C`:

```
from numpy import zeros
N = 1000
C = zeros([N,N],float)

for i in range(N):
    for j in range(N):
        for k in range(N):
            C[i,j] += A[i,k]*B[k,j]
```

We could use this code, for example, as the basis for a user-defined function to multiply arrays together. (As we saw in Section 2.4.4, Python already provides the function "`dot`" for calculating matrix products, but it's a useful exercise to write our own code for the calculation. Among other things, it helps us understand how many operations are involved in calculating such a product.)

How large a pair of matrices could we multiply together in this way if the calculation is to take a reasonable amount of time? The program has three nested for loops in it. The innermost loop, which runs through values of the variable `k`, goes around N times doing one multiplication operation each time and one addition, for a total of $2N$ operations. That whole loop is itself executed N times, once for each value of `j` in the middle loop, giving $2N^2$ operations. And those $2N^2$ operations are themselves performed N times as we go through the values of `i` in the outermost loop. The end result is that the matrix multiplication takes $2N^3$ operations overall. Thus if $N = 1000$, as above, the whole calculation would involve two billion operations, which is feasible in a few minutes of running time. Larger values of N, however, will rapidly become intractable. For $N = 2000$, for instance, we would have 16 billion op-

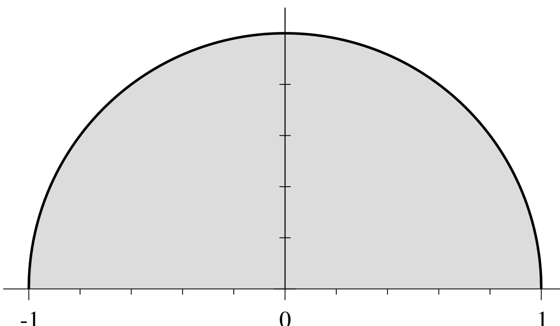
erations, which could take hours to complete. Thus the largest matrices we can multiply are about 1000×1000 in size.[4]

Exercise 4.4: Calculating integrals

Suppose we want to calculate the value of the integral

$$I = \int_{-1}^{1} \sqrt{1 - x^2}\, \mathrm{d}x.$$

The integrand looks like a semicircle of radius 1:



and hence the value of the integral—the area under the curve—must be equal to $\frac{1}{2}\pi = 1.57079632679\ldots$

Alternatively, we can evaluate the integral on the computer by dividing the domain of integration into a large number N of slices of width $h = 2/N$ each and then using the Riemann definition of the integral:

$$I = \lim_{N\to\infty} \sum_{k=1}^{N} h y_k \,,$$

---

[4]Interestingly, the direct matrix multiplication represented by the code given here is not the fastest way to multiply two matrices on a computer. *Strassen's algorithm* is an iterative method for multiplying matrices that uses some clever shortcuts to reduce the number of operations needed so that the total number is proportional to about $N^{2.8}$ rather than N^3. For very large matrices this can result in significantly faster computations. Unfortunately, Strassen's algorithm suffers from large numerical errors because of problems with subtraction of nearly equal numbers (see Section 4.2) and for this reason it is rarely used. On paper, an even faster method for matrix multiplication is the *Coppersmith–Winograd algorithm*, which requires a number of operations proportional to only about $N^{2.4}$, but in practice this method is so complex to program as to be essentially worthless—the extra complexity means that in real applications the method is always slower than direct multiplication.

where

$$y_k = \sqrt{1 - x_k^2} \qquad \text{and} \qquad x_k = -1 + hk.$$

We cannot in practice take the limit $N \to \infty$, but we can make a reasonable approximation by just making N large.

a) Write a program to evaluate the integral above with $N = 100$ and compare the result with the exact value. The two will not agree very well, because $N = 100$ is not a sufficiently large number of slices.

b) Increase the value of N to get a more accurate value for the integral. If we require that the program runs in about one second or less, how accurate a value can you get?

Evaluating integrals is a common task in computational physics calculations. We will study techniques for doing integrals in detail in the next chapter. As we will see, there are substantially quicker and more accurate methods than the simple one we have used here.