Chapter 4

Accuracy and speed

In the preceding chapters we have seen the basic elements of programming in Python: input and output, variables, functions, 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 have not covered. In later chapters of the book, for example, we will introduce specialized features for doing linear algebra and performing Fourier transforms. But for now we have the main components we need to start doing physics.

There is, however, one fundamental issue that we have not touched upon. Computers have limitations. They cannot store values 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 is crucial to understand those limitations, as well as methods for mitigating 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 examples 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 will 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 the calculation *underflows* and 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.[3] Be aware, however, that calculations with integers, even simple arithmetic operations, take longer the more digits there are, and can take a very long time if there are very many digits.

---

[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}.

[3] These observations apply to most integers in Python, including normal integer variables and integer elements of lists. An important exception, however, is integer arrays. For the sake of speed, the elements of an integer array are limited to numbers that can be stored in 32 bits (on Windows) or 64 bits (Mac and Linux), which implies an allowed range of $\pm 2.1 \times 10^9$ or $\pm 9.2 \times 10^{18}$ respectively. These are certainly large ranges, but they can be exceeded on occasion, in which case the computer will complain.

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 factorials 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. The standard level of precision, by international agreement, 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:	0.00000000000000002384626 . . .

The difference between the true value of a number and its value on the computer is called the *rounding error* or *round-off 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 you add 1.1 and 2.2 in Python, then obviously the answer should be 3.3, but the computer might give 3.299999999999999 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 have a statement like

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

because it may not do what you want. If the value of x is supposed to be 3.3 but it is actually 3.299999999999999, then as far as the computer is concerned it is 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 2.2.7, 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 do not have to be exactly equal. If `x` is 3.299999999999999 things will still work as expected. The value of `epsilon` has to be chosen appropriately for the situation—there is 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 by δ, is defined to be the difference between the true value and the value calculated by the computer. Or, equivalently, it is 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 not end up with exactly $x = \sqrt{2}$ (since $\sqrt{2}$ has an infinite number of decimal digits) but rather with $x + \delta = \sqrt{2}$, where δ is the rounding error, or equivalently $x = \sqrt{2} - \delta$. 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 13.79 ± 0.02 billion years, we mean that the measured value is 13.79 billion years, but the true value is possibly greater or less than this by an amount of order 0.02 billion years.

The error δ in the example above could be either positive or negative, depending on how the variable x gets rounded off. 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 is usually a good assumption to consider the error to be a (uniformly distributed) random number with standard deviation $\sigma = \epsilon x$, where $\epsilon \simeq 10^{-16}$ in this case. The quantity ϵ is called the *machine precision* or *machine epsilon* of the computer.[4] 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 δ itself, since we don't know it—if we did, then we could calculate $x + \delta$ and recover the exact value for the quantity of interest, so there would in effect be no error on 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. There are how-

[4]The technical definition of the machine epsilon is the difference between 1 and the smallest number greater than 1 that the computer can faithfully represent. Thus if we have 16 digits of precision then $\epsilon = 10^{-16}$.

ever certain situations in which problems can arise. One is when you are adding or subtracting numbers of very different sizes. If you are adding a series of numbers together and some are much smaller than others then the smaller ones can get lost. A simple way to mitigate this problem is to add the numbers together in order from smallest magnitude to largest (regardless of their signs). Doing this requires extra work to sort the numbers into the correct order, so it is only worthwhile if numerical precision is a significant issue for the particular calculation you are doing.

However, the most severe problems arise when the numbers are of closely *similar* size, and specifically when you are subtracting numbers. Suppose, for instance, that we have the following two numbers:

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

and suppose 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. In situations like this our accuracy becomes very poor indeed.

To put this in more general terms, if the difference between two numbers is small, so that it is comparable with the rounding error on the numbers, then the fractional error can become very large. In computer science this phenomenon is called *catastrophic cancellation*.

EXAMPLE 4.1: CATASTROPHIC CANCELLATION

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

$$x = 1, \qquad y = 1 + 10^{-14}\sqrt{2}. \tag{4.1}$$

Trivially we see that

$$10^{14}(y - x) = \sqrt{2}. \tag{4.2}$$

Let us perform the 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. (4.2) while the last line prints the true value of $\sqrt{2}$ (at least to the accuracy of the computer). Here is what we get when we run the program:

```
1.42108547152
1.41421356237
```

As we can see, the calculation of $10^{14}(y-x)$ 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 do not 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 is 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:

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

On the computer we cannot actually take the limit as h goes to zero, but we can get a reasonable approximation just by making h 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 $h = 10^{-2}$. 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 $h = 10^{-4}$, 10^{-6}, 10^{-8}, 10^{-10}, 10^{-12}, and 10^{-14}. You should see that the accuracy of the calculation initially gets better as h 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.

Exercise 4.4: Calculating variances

The mean and variance of a set of N numbers $x_1 \ldots x_N$ are given by the standard formulas

$$\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i, \qquad \operatorname{var} x = \frac{1}{N}\sum_{i=1}^{N} (x_i - \bar{x})^2. \tag{4.3}$$

The latter expression is often rewritten as

$$\operatorname{var} x = \frac{1}{N}\sum_{i=1}^{N} \bigl(x_i^2 - 2x_i\bar{x} + \bar{x}^2\bigr) = \overline{x^2} - \bar{x}^2, \tag{4.4}$$

where $\overline{x^2}$ is the mean-square value of x.

a) By any means you like, work out on paper the variance of the five numbers $x-2$, $x-1$, x, $x+1$, and $x+2$. You should find that the result is just a constant, independent of x.

b) Setting x to 1 billion, write a program to calculate the variance of these five numbers in two different ways, first using Eq. (4.3) and then using Eq. (4.4). What do you observe and how do you explain it?

c) In practice, if you were going to write a program to calculate the variance, which formula should you use?

4.3 PROGRAM SPEED

As we have seen, computers are not infinitely accurate. Neither are they infinitely fast. They work at amazing speeds, but many physics calculations require the computer to perform millions or billions of individual computations to get a desired final

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.

It will be useful to us to have a feel for how fast computers really are. As a general guide, performing a million mathematical operations is no big problem for a modern 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. A billion operations would take longer, typically a few minutes—less convenient, but still acceptable if you are patient. Performing a trillion operations, however, would take a few days. For very important or valuable results we may be willing to wait this long, but most people doing day-to-day computational physics would not. So a fair rule of thumb is that the calculations we can perform on a computer are ones that can be done in *a few 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 quick and easy) versus, say, calculating Bessel functions or multiplying matrices (which are not). Computers are also not all equal, and it makes a difference whether you are doing a calculation on a ten-year-old laptop or a supercomputer with a thousand CPUs. Moreover, computers are getting faster all the time and calculations may be within reach tomorrow that would be impossible today. Overall, however, limiting yourself to a few billion operations is a good general rule when thinking about what is possible in computational physics.

Example 4.2: The Madelung constant

In condensed matter physics the *Madelung constant* gives the total electric potential felt by an atom in a solid. It depends on the charges on the other atoms nearby and their locations. Consider for instance solid sodium chloride—table salt. The sodium chloride crystal has atoms arranged on a cubic lattice, but with alternating sodium and chlorine atoms, the sodium ones having a single positive charge $+e$ and the chlorine ones a single negative charge $-e$, where e is the charge on the electron. If we label each position in the lattice by three integer coordinates i, j, k, then the sodium atoms fall at positions where $i + j + k$ is even and the chlorine atoms at positions where $i + j + k$ is odd.

Consider a sodium atom at the origin $i = j = k = 0$. If the spacing of atoms in the lattice is a, then the distance from the origin to the atom at position (i, j, k) is

$$\sqrt{(ia)^2 + (ja)^2 + (ka)^2} = a\sqrt{i^2 + j^2 + k^2}, \tag{4.5}$$

and the potential at the origin created by such an atom is

$$V(i, j, k) = (-1)^{i+j+k} \frac{e/4\pi\epsilon_0}{a\sqrt{i^2 + j^2 + k^2}}, \tag{4.6}$$

with ϵ_0 being the permittivity of the vacuum and the factor of $(-1)^{i+j+k}$ giving us the appropriate positive or negative sign depending on whether $i + j + k$ is even or odd. The total potential felt by the sodium atom is then the sum of this quantity over all other atoms.

Let us assume a cubic sample of salt around the sodium at the origin, with L atoms in all directions. Then the total electric potential felt by the atom is

$$V_{\text{total}} = \sum_{\substack{i,j,k=-L \text{ to } L \\ \text{not } i=j=k=0}} V(i,j,k) = \frac{e}{4\pi\epsilon_0 a} M, \tag{4.7}$$

where the Madelung constant M is the dimensionless quantity

$$M = \sum_{\substack{i,j,k=-L \text{ to } L \\ \text{not } i=j=k=0}} \frac{(-1)^{i+j+k}}{\sqrt{i^2+j^2+k^2}}. \tag{4.8}$$

Technically the Madelung constant is actually the value when $L \to \infty$, but one can get an approximation just by making L large, although the quality of the approximation depends on how large a value we use. Of course, larger L also means the calculation will take longer to complete, because there are more terms in the sum, so there is a balance to be struck between accuracy and computation time. Let us see how good an answer we can get in a reasonable amount of time.

Here is a program to calculate the Madelung constant from the formula above:

```
from math import sqrt

L = 10
M = 0.0
for i in range(-L,L+1):
    for j in range(-L,L+1):
        for k in range(-L,L+1):
            if i==0 and j==0 and k==0:
                continue
            if (i+j+k)%2==0:
                M += 1/sqrt(i*i+j*j+k*k)
            else:
                M -= 1/sqrt(i*i+j*j+k*k)

print("M =",M)
```

Note a few features of this program:

1. We have three loops, nested inside one another, which run through every possible value of i, j, and k between $-L$ and L.
2. If $i = j = k = 0$ we skip the loop using the `continue` statement, since this term is excluded from the sum. Otherwise, we check whether $i + j + k$ is even or

odd (using the modulo operator) and add the appropriate term to the running total of M.

3. The total number of terms in the sum is $(2L+1)^3-1$ and each term involves five main operations—squaring the three coordinates, calculating the square root, and then taking the reciprocal of the result. (Also there are three additions, but these are generally quite quick and don't add much to the running time.) So the total number of operations is $5\times[(2L+1)^3-1]\simeq 40L^3$. With $L=10$ as above, for example, we have to do about 40 000 operations.

If we run the program we get this result:

```
M = -1.6925789282594415
```

This calculation takes 0.02 seconds on the author's laptop. Now let us try increasing the value of L. This will make our approximation more accurate and give us a better estimate of the Madelung constant, but at the expense of taking more time. If we increase L to 100, for instance, the answer changes quite significantly:

```
M = -1.7418198158396654
```

This calculation requires $40L^3 = 40$ million operations and takes 1.7 seconds, which is significantly longer than before, but still a short time in absolute terms. Normally we would be fine waiting this long to get a better answer.

But now let us increase L to 1000, meaning the number of operations is 40 billion. When we do this the calculation takes 29 minutes to finish, but the result changes only slightly:

```
M = -1.7469875326230075
```

There are three morals to this story. First, a few billion operations is indeed doable—if a calculation is important to us we can probably wait half an hour for an answer. But this is approaching the limit of what is reasonable. If we increased L by another factor of ten to $L = 10\,000$, the calculation would take a thousand times longer or about three weeks, which is not practical for most people.

Second, there is a balance to be struck between time spent and accuracy. In this case it was worthwhile to do the calculation with $L = 100$. It didn't take long and the result was noticeably improved from the result for $L = 10$. But it is arguable whether the change to $L = 1000$ was worth the effort—the calculation took much longer to complete but the answer was little changed. If we needed a particularly accurate answer, we might be willing to take the extra time to get it, but there are diminishing returns as we invest larger and larger amounts of time. We will see plenty of further examples in this book of calculations like this where we need to find an appropriate balance between speed and accuracy.

Third, it is pretty easy to write a program that will basically take forever to finish, so it is 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 is going to take a year then it is 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 more than a few billion, you may 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 matrices together. (As we saw in Section 2.4.4, Python already provides the function "`dot`" for calculating matrix products, but it is 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 plus one addition. Additions take a negligible amount of time compared to multiplications though, so it is safe to ignore them and just say we do a total of N operations each time around the innermost loop. That whole loop is itself executed N times, once for each value of `j` in the middle loop, giving N^2 operations. And those N^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 N^3 operations overall. Thus if $N = 1000$, as above, the whole calculation would involve a billion operations, which is feasible in a minute or two of running time. Larger values of N, however, will rapidly become intractable. For $N = 10\,000$, for instance, we would have a trillion operations, which could take hours or days to complete. Thus the largest matrices we can multiply in

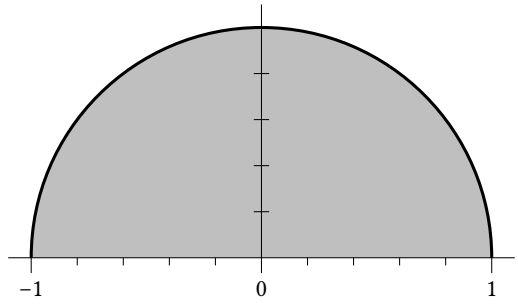
reasonable time are about 1000×1000 in size or a little larger.[5]

Exercise 4.5: 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 $\frac{1}{2}\pi = 1.57079632679\ldots$

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:

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

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 just by making N large.

---

[5]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 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.

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 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 Chapter 5. As we will see, there are substantially quicker and more accurate methods than the simple one we have used here.

Chapter summary

- There is a limit to the largest and smallest values that can be represented by most Python variables. Floating-point variables have a maximum value of around 10^{308}. If you exceed this limit the calculation will **overflow** and the value of the variable becomes `inf`, meaning infinity.
- Integer variables normally have no largest value. They can become as large as you like, although calculations with very large integers are slow.
- Integer values stored specifically in arrays do have a largest value, which is either 2^{32} or 2^{64} depending on your operating system.
- Floating-point arithmetic is limited in its precision. In Python, floating-point values are normally stored to about 16 significant figures. Any numbers with more than 16 figures are rounded off, and this leads to **rounding error**.
- The size of the rounding error on a number x is given by ϵx, where ϵ is the **machine precision** or **machine epsilon**, which has a value of about $\epsilon \simeq 10^{-16}$ in Python.
- In many cases rounding errors are small enough that you can ignore them, but there are three particular situations where they may not be. The first occurs when you are testing two floats to see if they are equal. The second is when you are adding or subtracting numbers of very different sizes, in which case smaller numbers can get washed out by larger ones. The third and most important situation is when you are subtracting two numbers with closely equal values, which can lead to the phenomenon of **catastrophic cancellation**.
- Computers are fast, but not infinitely fast. On today's computers a Python program can perform a few billion mathematical operations in a few minutes. This sets a rough limit for what can be done in reasonable time. The calculations that can be done on a computer are ones that involve **a few billion operations or less.**