2 Numbers

An educated mind expects only the precision which a given subject admits.

Aristotle

2.1 Motivation

We don't really have to provide a motivation regarding the importance of numbers in physics: both experimental measurements and theoretical calculations produce specific numbers. Preferably, one also estimates the experimental or theoretical uncertainty associated with these values.

We have, semi-arbitrarily, chosen to discuss a staple of undergrad physics education, the photoelectric effect. This arises when a metal surface is illuminated with electromagnetic radiation of a given frequency ν and electrons come out. In 1905 Einstein posited that quantization is a feature of the radiation itself. Thus, a light quantum (a *photon*) gives up its energy ($h\nu$, where h is now known as *Planck's constant*) to an electron, which has to break free of the material, coming out with a kinetic energy of:

$$E = h\nu - W \tag{2.1}$$

where W is the work function that relates to the specific material. The maximum kinetic energy E of the photoelectrons could be extracted from the potential energy of the electric field needed to stop them, via $E = eV_s$, where V_s is the stopping potential and e is the charge of the electron. Together, these two relations give us:

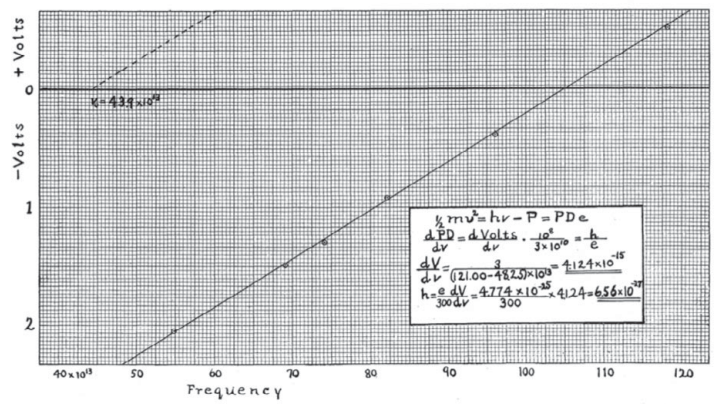
$$eV_s = h\nu - W \tag{2.2}$$

Thus, if one produces data relating V_s with ν, the slope would give us h/e.

In 1916, R. A. Millikan published a paper [67] titled "A direct photoelectric determination of Planck's h", where he did precisely that. Millikan's device included a remotely controlled knife that would shave a thin surface off the metal; this led to considerably enhanced photocurrents. It's easy to see why Millikan described his entire experimental setup as a "machine shop in vacuo". Results from this paper are shown in Fig. 2.1. Having extracted the slope h/e, the author then proceeded to compute h by "inserting my value of e".[1] The value Millikan extracted for Planck's constant was:

$$h = 6.56 \times 10^{-27} \text{erg s} \tag{2.3}$$

[1] If you are surprised by this turn of phrase, you should recall that Millikan had measured e very accurately with his oil-drop experiment in 1913.



Millikan's data on the photoelectric effect. Reprinted figure with permission from R. A. Millikan, *Phys. Rev.* **7**, 355 (1916), Copyright 1916 by the American Physical Society. **Fig. 2.1**

In his discussion of Fig. 2.1, Millikan stated that "it is a conservative estimate to place the maximum uncertainty in the slope at about 1 in 200 or 0.5 per cent".[2] Translating this to the above units, we find an uncertainty estimate of 0.03×10^{-27}erg s. It's worth noting that Richardson and Compton's earlier experimental results [78] had a slope uncertainty that was larger than 60 percent. The above extraction is to be compared with the modern determination of $h = 6.626070040(81) \times 10^{-27}$erg s. Overall, Millikan's result was a huge improvement on earlier works and turned out to be important in the acceptance of Einstein's work and of light quanta.[3]

For the sake of completeness, we note that the aforementioned error estimate follows from *assuming* a linear relationship. It would have probably been best to start from Millikan's earlier comment that "the maximum possible error in locating any of the intercepts is say two hundredths of a volt" and then do a least-squares fit, as explained in chapter 6. This isolated example already serves to highlight that even individual experimental measurements have associated uncertainties; nowadays, experimental data points are always given along with a corresponding error bar. Instead of multiplying the examples where experiment and theory interact fruitfully, we now turn to the main theme of this chapter, which is the presence of errors when storing and computing numbers.

2.2 Errors

In this text we use the word *accuracy* to describe the match of a value with the (possibly unknown) true value. On the other hand, we use the word *precision* to denote how many

[2] Reprinted excerpts with permission from R. A. Millikan, *Phys. Rev.* **7**, 355 (1916), Copyright 1916 by the American Physical Society.

[3] Intriguingly, Millikan himself was far from being convinced that quantum theory was relevant here, speaking of "the bold, not to say the reckless, hypothesis of an electro-magnetic light corpuscle".

digits we can use in a mathematical operation, whether these digits are correct or not. An inaccurate result arises when we have an error. This can happen for a variety of reasons, only one of which is limited precision. Excluding "human error" and measurement uncertainty in the input data, there are typically two types of errors we have to deal with in numerical computing: approximation error and rounding error. In more detail:

- **Approximation errors** Here's an example. You are trying to approximate the exponential, e^x, using its Taylor series:

$$y = \sum_{n=0}^{n_{max}} \frac{x^n}{n!} \tag{2.4}$$

  Obviously, we are limiting the sum to the terms up to n_{max} (i.e., we are including the terms labelled 0, 1, ..., n_{max} and dropping the terms from $n_{max} + 1$ to ∞). As a result, it's fairly obvious that the value of y for a given x may depend on n_{max}. In principle, at the mere cost of running one's calculation longer, one can get a better answer.[4]
- **Roundoff errors** These are also known as *rounding errors*. This type of error appears every time a calculation is carried out using floating-point numbers: since these don't have infinite precision, some information is lost. Here's an example: using real numbers, it is easy to see that $(\sqrt{2})^2 - 2 = 0$. However, when carrying out the same operation in Python we get a non-zero answer:

```
>>> (sqrt(2))**2 - 2
4.440892098500626e-16
```

  This is because $\sqrt{2}$ cannot be evaluated with infinitely many digits on the computer. Thus, the (slightly inaccurate) result for `sqrt(2)` is then used to carry out a second calculation, namely the squaring. Finally, the subtraction is yet another mathematical operation that can lead to rounding error.[5] Often, roundoff errors do not go away even if you run the calculation longer.

In the present chapter we will talk quite a bit about roundoff errors. In the next chapter we talk about the combined effect of approximation and roundoff errors. The chapters after that typically focus only on approximation errors, i.e., on estimating how well a specific method performs in principle. Before we get that far, however, let us first try to introduce some basic concepts, without limiting ourselves to any one kind of error.

2.2.1 Absolute and Relative Error

Assume we are studying a quantity whose exact value is x. If $\tilde{x}$ is an approximate value for it, then we can define the *absolute error* as follows:[6]

[4] But keep reading, since roundoff error becomes important here, too.
[5] Again, this is discussed much more thoroughly in the rest of the chapter.
[6] Other authors employ a different definition of the absolute error, which differs by an overall minus sign.

$$\Delta x = \tilde{x} - x \tag{2.5}$$

We don't specify at this point the source of this absolute error: it could be uncertainties in the input data, an inaccuracy introduced by our imperfect earlier calculation, or the result of roundoff error (possibly accumulated over several computations). For example:

$$x_0 = 1.000, \qquad \tilde{x}_0 = 0.999 \tag{2.6}$$

corresponds to an absolute error of $\Delta x_0 = -10^{-3}$. This also allows us to see that the absolute error, as defined, can be either positive or negative. If you need it to be positive (say, in order to take its logarithm), simply take the absolute value.

We are usually interested in defining an *error bound* of the form:

$$|\Delta x| \leq \epsilon \tag{2.7}$$

or, equivalently:

$$|\tilde{x} - x| \leq \epsilon \tag{2.8}$$

where we hope that ϵ is "small". Having access to such an error bound means that we can state something very specific regarding the (unknown) exact value x:

$$\tilde{x} - \epsilon \leq x \leq \tilde{x} + \epsilon \tag{2.9}$$

This means that, even though we don't know the exact value x, we do know that it could be at most $\tilde{x} + \epsilon$ and at the least $\tilde{x} - \epsilon$. Keep in mind that if you know the actual absolute error, as in our $\Delta x_0 = -10^{-3}$ example above, then, from Eq. (2.5), you know that:

$$x = \tilde{x} - \Delta x \tag{2.10}$$

and there's no need for inequalities. The inequalities come into the picture when you don't know the actual value of the absolute error and only know a bound for the magnitude of the error. The error bound notation $|\Delta x| \leq \epsilon$ is sometimes rewritten in the form $x = \tilde{x} \pm \epsilon$, though you should be careful: this employs our definition of *maximal error* (i.e., the worst-case scenario) as above, not the usual *standard error* (i.e., the statistical concept you may have encountered in a lab course).

Of course, even at this early stage, one should think about exactly what we mean by "small". Our earlier case of $\Delta x_0 = -10^{-3}$ probably fits the bill. But what about:

$$x_1 = 1\,000\,000\,000.0, \qquad \tilde{x}_1 = 999\,999\,999.0 \tag{2.11}$$

which corresponds to an absolute error of $\Delta x_1 = -1$? Obviously, this absolute error is larger (in magnitude) than $\Delta x_0 = -10^{-3}$. On the other hand, it's not too far-fetched to say that there's something wrong with this comparison: x_1 is much larger (in magnitude) than x_0, so even though our approximate value $\tilde{x}_1$ is off by a unit, it "feels" closer to the corresponding exact value than $\tilde{x}_0$ was.

This is resolved by introducing a new definition. As before, we are interested in a quantity whose exact value is x and an approximate value for it is $\tilde{x}$. Assuming $x \neq 0$, we can define the *relative error* as follows:

$$\delta x = \frac{\Delta x}{x} = \frac{\tilde{x} - x}{x} \tag{2.12}$$

Obviously, this is simply the absolute error Δx divided by the exact value x. As before, we are not specifying the source of this relative error (input-data uncertainties, roundoff, etc.). Another way to express the relative error is:

$$\tilde{x} = x(1 + \delta x) \tag{2.13}$$

You should convince yourself that this directly follows from Eq. (2.12). We will use this formulation repeatedly in what follows.[7]

Let's apply our definition of the relative error to the earlier examples:

$$\delta x_0 = \frac{0.999 - 1.000}{1.000} = -10^{-3}, \qquad \delta x_1 = \frac{999\,999\,999.0 - 1\,000\,000\,000.0}{1\,000\,000\,000.0} = -10^{-9} \tag{2.14}$$

The definition of the relative error is consistent with our intuition: $\tilde{x}_1$ is, indeed, a much better estimate of x_1 than $\tilde{x}_0$ is of x_0. Quite frequently, the relative error is given as a percentage: δx_0 is a relative error of -0.1% whereas δx_1 is a relative error of $-10^{-7}\%$.

In physics the values of an observable can vary by several orders of magnitude (according to density, temperature, and so on), so it is wise to employ the scale-independent concept of the relative error, when possible.[8] Just like for the case of the absolute error, we can also introduce a *bound for the relative error*:

$$|\delta x| = \left|\frac{\Delta x}{x}\right| \leq \epsilon \tag{2.15}$$

where now the phrase "ϵ is small" is unambiguous (since it doesn't depend on whether or not x is large). Finally, we note that the definiton of the relative error in Eq. (2.12) involves x in the denominator. If we have access to the exact value (as in our examples with x_0 and x_1 above), all is well. However, if we don't actually know the exact value, it is sometimes more convenient to use, instead, the approximate value $\tilde{x}$ in the denominator. A problem discusses this alternative definition and its connection with what we discussed above.

2.2.2 Error Propagation

So far, we have examined the concepts of the absolute error and of the relative error (as well as the corresponding error bounds). These were discussed in general, without any

[7] You can also expand the parentheses and identify $x\delta x = \Delta x$, to see that this leads to $\tilde{x} = x + \Delta x$.
[8] Once again, if you need the relative error to be positive, simply take the absolute value.

details provided regarding the mathematical operations carried out using these numbers. We now proceed to discuss the elementary operations (addition, subtraction, multiplication, division), which will hopefully give you some insights into combining approximate values together. One of our goals is to see what happens when we put together the error bounds for two numbers a and b to produce an error bound for a third number, x. In other words, we will study error propagation. Near the end of the section, we will study more general scenarios, in which one is faced with more complicated mathematical operations (and possibly more than two numbers being operated on). In what follows, it's important to keep in mind that these are *maximal errors*, so our results will be different (and likely more pessimistic) than what you may have encountered in a standard course on experimental measurements.

Addition or Subtraction

We are faced with two real numbers a and b, and wish to take their difference:

$$x = a - b \tag{2.16}$$

As usual, we don't know the exact values, but only the approximate values $\tilde{a}$ and $\tilde{b}$, so what we form instead is the difference of these:

$$\tilde{x} = \tilde{a} - \tilde{b} \tag{2.17}$$

Let us now apply Eq. (2.10) twice:

$$\tilde{a} = a + \Delta a, \qquad \tilde{b} = b + \Delta b \tag{2.18}$$

Plugging the last four equations into the definition of the absolute error, Eq. (2.5), we have:

$$\Delta x = \tilde{x} - x = (a + \Delta a) - (b + \Delta b) - (a - b) = \Delta a - \Delta b \tag{2.19}$$

In the third equality we cancelled what we could.

We now recall that we are interested in finding relations between error bounds. Thus, we take the absolute value and then use the triangle inequality to find:

$$|\Delta x| \leq |\Delta a| + |\Delta b| \tag{2.20}$$

You should convince yourself that a fully analogous derivation leads to exactly the same result for the case of addition of the two numbers a and b. Thus, our main conclusion so far is that *in addition and subtraction adding together the bounds for the absolute errors in the two numbers gives us the bound for the absolute error in the result*.

Let's look at an example. Assume that we have:

$$|4.56 - a| \leq 0.14, \qquad |1.23 - b| \leq 0.03 \tag{2.21}$$

(If you are in any way confused by this notation, look up Eq. (2.7) or Eq. (2.8).) Our finding in Eq. (2.20) implies that the following relation will hold, when $x = a - b$:

$$|3.33 - x| \leq 0.17 \tag{2.22}$$

It's easy to see that this error bound, simply the sum of the two error bounds we started with, is larger than either of them. If we didn't have access to Eq. (2.20), we could have arrived at the same result the long way: (a) when a has the greatest possible value (4.70) and b has the smallest possible value (1.20), we get the greatest possible value for $a - b$, namely 3.50, and (b) when a has the smallest possible value (4.42) and b has the greatest possible value (1.26), we get the smallest possible value for $a - b$, namely 3.16.

As the form of our main result in Eq. (2.20), applied just now in a specific example, shows, we simply add up the absolute error bounds. As mentioned earlier, this is different from what you do when you are faced with "standard errrors" (namely, standard deviations of the sampling distribution): in that case, the absolute errors add "in quadrature". We repeat that our result is more pessimistic (i.e., tries to account for the worst-case scenario). We won't keep repeating this warning below.

Catastrophic Cancellation

Let us examine the most interesting special case: $a \approx b$ (for which case $x = a - b$ is small). Dividing our result in Eq. (2.20) with x gives us the relative error (bound) in x:

$$|\delta x| = \left|\frac{\Delta x}{x}\right| \leq \frac{|\Delta a| + |\Delta b|}{|a - b|} \tag{2.23}$$

Now, express Δa and Δb in terms of the corresponding relative error: $\Delta a = a\delta a$ and $\Delta b = b\delta b$. Since $a \approx b$, you can factor $|a|$ out:

$$|\delta x| \leq (|\delta a| + |\delta b|)\frac{|a|}{|a - b|} \tag{2.24}$$

It's easy to see that if $a \approx b$ then $|a - b|$ will be much smaller than $|a|$ so, since the fraction will be large, the relative errors δa and δb will be magnified.[9]

Let's look at an example. Assume that we have:

$$|1.25 - a| \leq 0.03, \qquad |1.20 - b| \leq 0.03 \tag{2.25}$$

that is, a relative error (bound) $\delta a \approx 0.03/1.25 = 0.024$ or roughly 2.4% (this is approximate, because we divided with $\tilde{a}$, not with the, unknown, a). Similarly, the other relative error (bound) is $\delta b \approx 0.03/1.20 = 0.025$ or roughly 2.5%. From Eq. (2.24) we see that the relative error for the difference will obey:

$$|\delta x| \leq (0.024 + 0.025)\frac{1.25}{0.05} = 1.225 \tag{2.26}$$

where the right-hand side is an approximation (using $\tilde{a}$ and $\tilde{x}$). This shows us that two numbers with roughly 2.5% relative errors were subtracted and the result has a relative error which is more than one hundred percent! This is sometimes known as *subtractive or catastrophic cancellation*. For the purists, we note that *catastrophic cancellation* refers to the case where the two numbers we are subtracting are themselves subject to errors, as

[9] This specific issue doesn't arise in the case of addition, since there the denominator doesn't have to be tiny.

above. There also exists the scenario of *benign cancellation* which shows up when you subtract quantities that are exactly known (though that is rarely the case in practice) or when the result of the subtraction does not need to be too accurate for what follows. The distinction between catastrophic and benign cancellation is further explained in the problems (including the classic example of a simple quadratic equation) and in section 2.4 below.

Multiplication or Division

We are faced with two real numbers a and b, and wish to take their product:

$$x = ab \tag{2.27}$$

As usual, we don't know the exact values, but only the approximate values $\tilde{a}$ and $\tilde{b}$, so what we form instead is the product of these:

$$\tilde{x} = \tilde{a}\tilde{b} \tag{2.28}$$

With some foresight, we will now apply Eq. (2.13) twice:

$$\tilde{a} = a(1 + \delta a), \qquad \tilde{b} = b(1 + \delta b) \tag{2.29}$$

Plugging the last four equations into the definition of the relative error, Eq. (2.12), we have:

$$\delta x = \frac{\tilde{x} - x}{x} = \frac{\tilde{a}\tilde{b} - ab}{ab} = \frac{a(1 + \delta a)b(1 + \delta b) - ab}{ab} = 1 + \delta a + \delta b + \delta a \delta b - 1 = \delta a + \delta b \tag{2.30}$$

In the fourth equality we cancelled the ab. In the fifth equality we cancelled the unit and we dropped $\delta a \delta b$ since this is a higher-order term (it is the product of two small terms).

We now recall that we are interested in finding relations between error bounds. Thus, we take the absolute value and then use the triangle inequality to find:

$$|\delta x| \leq |\delta a| + |\delta b| \tag{2.31}$$

In one of the problems you will carry out the analogous derivation for the case of division of the two numbers a and b, finding exactly the same result. Thus, our new conclusion is that *in multiplication and division adding together the bounds for the relative errors in the two numbers gives us the bound for the relative error in the result*. Observe that for addition or subtraction we were summing *absolute* error bounds, whereas here we are summing *relative* error bounds. This distinction between the two is the source of the conclusion that typically multiplication and division don't cause too much trouble, whereas addition and (especially) subtraction can cause headaches.

Let's look at an example. Assume that we have the same numbers as in the section on catastrophic cancellation:

$$|1.25 - a| \leq 0.03, \qquad |1.20 - b| \leq 0.03 \tag{2.32}$$

that is, a relative error $\delta a \approx 0.03/1.25 = 0.024$ of roughly 2.4% and a relative error $\delta b \approx 0.03/1.20 = 0.025$ of roughly 2.5%. Our finding in Eq. (2.31) implies that:

$$\frac{|1.5 - x|}{|x|} \leq 0.049 \tag{2.33}$$

namely a relative error bound of roughly 4.9%. It's easy to see that this error bound, while larger than either of the relative error bounds we started with, is nowhere near as dramatic as what we found when we subtracted the same two numbers.

General Error Propagation: Functions of One Variable

We have studied (maximal) error propagation for a couple of simple cases of combining two numbers (subtraction and multiplication). But what about the error when you do something more complicated to a single number, e.g., take its square root or its logarithm?

Let us go over some notation. The absolute error in a variable x is defined as per Eq. (2.5):

$$\Delta x = \tilde{x} - x \tag{2.34}$$

We are now interested in a more involved quantity, namely $y = f(x)$. We start from the absolute error. We wish to calculate:

$$\Delta y = \tilde{y} - y = f(\tilde{x}) - f(x) \tag{2.35}$$

What we'll do is to Taylor expand $f(\tilde{x})$ around x. This gives us:

$$\begin{aligned} \Delta y &= f(x + \Delta x) - f(x) \\ &= f(x) + \frac{df(x)}{dx}(\tilde{x} - x) + \frac{1}{2}\frac{d^2 f(x)}{dx^2}(\tilde{x} - x)^2 + \cdots - f(x) \\ &\approx \frac{df(x)}{dx}(\tilde{x} - x) \end{aligned} \tag{2.36}$$

In the third line we cancelled the $f(x)$ and disregarded the $(\tilde{x} - x)^2$ term and higher-order contributions: assuming $\tilde{x} - x$ is small, this is legitimate. We now identify the term in parentheses from Eq. (2.34), leading to:

$$\Delta y \approx \frac{df(x)}{dx}\Delta x \tag{2.37}$$

In words, we see that $df(x)/dx$ determines how strongly the absolute error in x will affect the absolute error in y. To be specific, if you were faced with, say, $y = x^3$, then you could estimate the absolute error in y simply by taking a derivative: $\Delta y \approx 3x^2 \Delta x$. Obviously, when x is large the absolute error Δx gets amplified due to the presence of $3x^2$ in front.

It is straightforward to use Eq. (2.37) to get an estimate for the relative error in y:

$$\delta y = \frac{\Delta y}{y} \approx \frac{1}{f(x)}\frac{df(x)}{dx}\Delta x \tag{2.38}$$

If we now multiply by and divide with x (assuming, of course, that $x \neq 0$) we find:

$$\delta y \approx \frac{x}{f(x)} \frac{df(x)}{dx} \delta x \tag{2.39}$$

which is our desired relation connecting δy with δx. Analogously to what we saw for the case of the absolute error, our finding here shows us that the coefficient in front of δx determines how strongly the relative error in x will affect the relative error in y. For example, if you were faced with $y = x^4$, then you could estimate the relative error in y as follows:

$$\delta y \approx \frac{x}{x^4} 4x^3 \delta x = 4\delta x \tag{2.40}$$

that is, the relative error in y is worse than the relative error in x, but not dramatically so.[10]

General Error Propagation: Functions of Many Variables

For future reference, we oberve that it is reasonably straightforward to generalize our approach above to the case of a function of many variables, i.e., $y = f(x_0, x_1, \ldots, x_{n-1})$: the total error Δy would then have contributions from each Δx_i and each partial derivative:

$$\Delta y \approx \sum_{i=0}^{n-1} \frac{\partial f}{\partial x_i} \Delta x_i \tag{2.41}$$

This is a general formula, which can be applied to functions of varying complexity. As a trivial check, we consider the case:

$$y = x_0 - x_1 \tag{2.42}$$

which a moment's consideration will convince you is nothing other than the subtraction of two numbers, as per Eq. (2.16). Applying our new general result to this simple case:

$$\Delta y \approx \Delta x_0 - \Delta x_1 \tag{2.43}$$

which is precisely the result we found in Eq. (2.19).

Equation (2.41) can now be used to produce a relationship for the relative error:

$$\delta y \approx \sum_{i=0}^{n-1} \frac{x_i}{f(x_0, x_1, \ldots, x_{n-1})} \frac{\partial f}{\partial x_i} \delta x_i \tag{2.44}$$

You should be able to see that this formula can be applied to almost all possible scenarios.[11] An elementary test would be to take:

$$y = x_0 x_1 \tag{2.45}$$

[10] You will apply both Eq. (2.37) and Eq. (2.39) to other functions when you solve the relevant problem.

[11] Of course, in deriving our last result we assumed that $y \neq 0$ and that $x_i \neq 0$.

which is simply the multiplication of two numbers, as per Eq. (2.27). Applying our new general result for the relative error to this simple case, we find:

$$\delta y \approx \frac{x_0}{x_0 x_1} x_1 \delta x_0 + \frac{x_1}{x_0 x_1} x_0 \delta x_1 = \delta x_0 + \delta x_1 \tag{2.46}$$

which is precisely the result in Eq. (2.30).

You will benefit from trying out more complicated cases, e.g., $y = \sqrt{x_0} + x_1^3 - \log(x_2)$. In your resulting expression for δy, the coefficient in front of each δx_i tells you by how much (or if) the corresponding relative error is amplified.

2.3 Representing Real Numbers

Our discussion so far has been general: the source of the error, leading to an absolute or relative error or error bound, has not been specified. At this point, we will turn our attention to roundoff errors. In order to do that, we first go over the representation of real numbers on the computer and then discuss simple examples of mathematical operations. For the rest of the chapter, we will work on roundoff error alone: this will result from the representation of a number (i.e., storing that number) or the representation of an operation (e.g., carrying out a subtraction).

2.3.1 Basics

Computers use electrical circuits, which communicate using signals. The simplest such signals are *on* and *off*. These two possibilities are encoded in what is known as a *binary digit or bit*: bits can take on only two possible values, by convention 0 or 1.[12] All types of numbers are stored in binary form, i.e., as collections of 0s and 1s.

Python integers actually have unlimited precision, so we won't have to worry about them too much. In this book, we mainly deal with real numbers, so let's briefly see how those are represented on the computer. Most commonly, real numbers are stored using *floating-point representation*. This has the general form:

$$\pm \text{ mantissa} \times 10^{\text{exponent}} \tag{2.47}$$

For example, the speed of light in scientific notation is $+2.997\,924\,58 \times 10^8$ m/s.

Computers only store a finite number of bits, so cannot store exactly all possible real numbers. In other words, there are "only" finitely many exact representations/*machine numbers*.[13] These come in two varieties: normal and subnormal numbers. There are three ways of losing precision, as shown qualitatively in Fig. 2.2: *underflow* for very small numbers, *overflow* for very large numbers, and *rounding* for decimal numbers whose value falls

[12] You can contrast this to decimal digits, which can take on 10 different values, from 0 to 9.
[13] That is, finitely many decimal numbers that can be stored exactly using a floating-point representation.

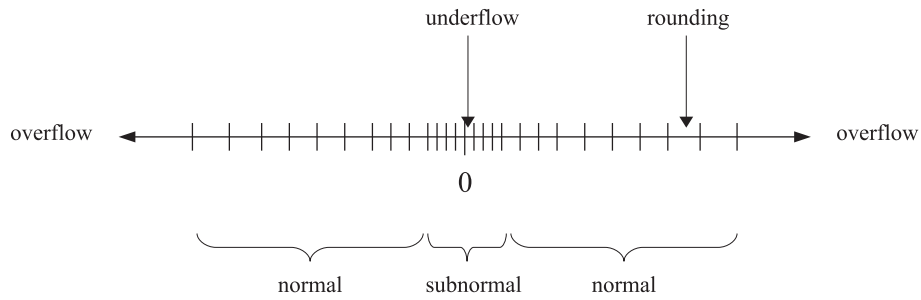


Illustration of exactly representable floating-point numbers

Fig. 2.2

between two exactly representable numbers. For more on these topics, you should look at Appendix B; here we limit ourselves to simply quoting some results.

Python employs what are known as *double-precision floating point numbers*, also called *doubles*; their storage uses 64 bits in total. Doubles can represent:

$$\pm\, 4.9 \times 10^{-324} \leftrightarrow \pm 1.8 \times 10^{308} \tag{2.48}$$

This refers to the ability to store very large or very small numbers. Most of this ability is found in the term corresponding to the exponent. For doubles, if we try to represent a number that's larger than 1.8×10^{308} we get *overflow*. Similarly, if we try to represent a number that's smaller than 4.9×10^{-324} we get *underflow*. Keep in mind that being able to represent 4.9×10^{-324} does *not* mean that we are able to store 324 significant figures in a double. The number of significant figures (and the related concept of *precision*) is found in the coefficient in front (e.g., `1.8` or `1.234567`). For doubles, the precision is 1 part in $2^{52} \approx 2.2 \times 10^{-16}$, which amounts to 15 or 16 decimal digits.

2.3.2 Overflow

We can explore the above results programmatically. We will start from an appropriately large value, in order to shorten the number of output lines:

```
>>> large = 2.**1021
for i in range(3):
...       large *= 2
...       print(i, large)
0 4.49423283715579e+307
1 8.98846567431158e+307
2 inf
```

This is what we expected: from $2^{1024} \approx 1.7976 \times 10^{308}$ and onward we are no longer able to store the result in a double. You can check this by saying:

```
>>> 8.98846567431158e+307*1.999999999999999
1.797693134862315e+308
```

Multiplying by 2 would have led to `inf`, as above. Explicitly seeing when we get underflow is left as a problem.

2.3.3 Machine Precision

We already mentioned that the precision for doubles is limited to $\approx 2.2 \times 10^{-16}$. The precision is related to the distance between two vertical lines in the figure above for a given region of interest: as we noted, anything between the two lines gets rounded, either to the left or to the right. We now turn to the question of carrying out arithmetic operations using such numbers: this gives rise to the all-important question of *rounding*. This question arises every time we are trying to combine two floating-point numbers but the answer is not an exactly representable floating-point number. For example, 1 and 10 can be exactly represented as doubles, but $1/10$ cannot.

We first address an even simpler problem: five-digit decimal arithmetic. Let's assume we want to add together the two numbers 0.12345 and 1.2345. One could notice that $1.2345 = 0.12345 \times 10^1$ while $0.12345 = 0.12345 \times 10^0$, i.e., in an (imagined) system that used five-digit decimal arithmetic these two numbers would have the same mantissa and different exponents. However, that doesn't help us when adding: to add two mantissas we have to be sure they correspond to the same exponent (since that's how addition works). Adding them as real numbers (i.e., not five-digit decimal numbers) we get:

$$0.12345 + 1.2345 = 1.35795 \tag{2.49}$$

But our answer now contains six decimal digits, 1.35795. Since we're limited to five-digit decimal numbers, this leaves us with the option of *chopping* the result down to 1.3579 or *rounding* it up to 1.3580. Problems like this one also appear in other arithmetic operations and for other representational systems (like binary).

Turning back to the question of the machine representation of doubles, we try to make the concept of "precision" more specific. We define the *machine precision* ϵ_m as follows: it is the gap between the number 1.0, on the one hand, and the smallest possible number $\tilde{x}$ that is larger than 1.0 ($\tilde{x} > 1.0$), on the other hand. If you have read Appendix B, you should know that (given the form of the mantissa for normal doubles) the smallest number we can represent obeying $\tilde{x} > 1.0$ is $1 + 2^{-52}$. The gap between this number and 1 is $2^{-52} \approx 2.2 \times 10^{-16}$. In other words:

$$\epsilon_m \approx 2.2 \times 10^{-16} \tag{2.50}$$

We will make use of this repeatedly in what follows.

Instead of delving into a study of binary arithmetic (analogous to what we did above for five-digit decimal arithmetic), let us investigate the definition of machine precision using

Python. We start with a small number and keep halving it, after which operation we add it on to `1.0`: at some point, this is going to give us back `1.0`: we then call the gap between `1.0` and the last number > 1 the machine precision. Explicitly:

```
>>> small = 1/2**50
>>> for i in range(3):
...       small /= 2
...       print(i, 1 + small, small)
0 1.0000000000000004 4.440892098500626e-16
1 1.0000000000000002 2.220446049250313e-16
2 1.0 1.1102230246251565e-16
```

As you can see, we started `small` at an appropriately small value, in order to shorten the number of output lines. We can further explore this topic interactively. At first sight, the results below might be confusing:

```
>>> 1. + 2.3e-16
1.0000000000000002
>>> 1. + 1.6e-16
1.0000000000000002
>>> 1. + 1.12e-16
1.0000000000000002
>>> 1. + 1.1e-16
1.0
```

We found that there exist numbers smaller than `2.2e-16` that when added to 1 lead to a result that is larger than 1. If you pay close attention, you will notice that `1. + 1.6e-16` or `1. + 1.12e-16` are rounded to the same number that `1. + 2.22e-16` corresponds to (namely, `1.0000000000000002`). However, below a certain point, we start rounding down to 1. In other words, for some values of `small` below the machine precision, we have a computed value of `1.+small` that is not 1, but corresponds to $1 + \epsilon_m$.[14]

Take a moment to appreciate that you can use a double to store a tiny number like 10^{-300}, but this doesn't mean that you can store $1 + 10^{-300}$: to do so, you would need 301 digits of precision (and all you have is 16).

2.3.4 Revisiting Subtraction

We discussed in an earlier section how bad catastrophic cancellation can be. At the time, we were investigating general errors, which could have come from several sources. Let us try to specifically investigate what happens in the case of subtraction when the *only* errors involved are those due to roundoff, i.e., let us assume that the relative error δx is a consequence of the fact that, generally speaking, x cannot be represented exactly on the

[14] Some authors call the smallest value of `small` for which `1.+small` doesn't round down to 1 the *unit roundoff* u: it is easy to see that this is related to the machine precision by $\epsilon_m = 2u$.

computer (i.e., it is typically not a machine number). Thus, when replacing x by its nearest double-precision floating-point number, we make a roundoff error: without getting into more detail, we will estimate the error's magnitude using the machine precision, which as we saw earlier is $\epsilon_m \approx 2.2 \times 10^{-16}$.[15] Obviously, this is only an estimate: for example, the error made when rounding to a subnormal number may be smaller (since subnormals are more closely spaced). Another example: if x is a machine number, then it can be exactly represented by $\tilde{x}$, so the relative error is actually 0. Our point in using ϵ_m is simply that one should typically not trust floating-point numbers for more than 15–16 (relative) decimal digits of precision.[16] If you think of ϵ_m as an error bound, then the fact that sometimes the error is smaller is OK.

Since we will estimate the relative error δx using ϵ_m, we see that the absolute error can be considerably larger: from Eq. (2.12) we know that $\Delta x = x\delta x$, so if x is very large then since we are taking $\epsilon_m \approx 2.2 \times 10^{-16}$ as being fixed at that value, it's obvious that Δx can become quite large. Let's take a specific example: given a relative error $\approx 10^{-16}$, a specific double of magnitude $\approx 10^{22}$ will have an absolute error in its last digit, of order 10^6.

Given that we use ϵ_m to find a general relative error in representing a real number via the use of a floating-point number, we can re-express our result for catastrophic cancellation from Eq. (2.24) as:

$$|\delta x| \leq \frac{|a|}{|a-b|}\, 2\epsilon_m \tag{2.51}$$

Thus, due to the presence of $|a|/|a-b|$, the relative error when subtracting two nearly equal numbers can be much larger than ϵ_m.

It might be worth making a distinction at this point between: (a) the loss of significant figures when subtracting two nearly equal numbers, and (b) the loss of even more digits when carrying out the subtraction using floating-point numbers (which have finite precision). Let's start from the first case. Subtract two nearly equal numbers, each of which has 20 significant figures:

$$1.2345678912345678912 - 1.2345678900000000000 = 0.0000000012345678912 \tag{2.52}$$

Note that here we subtracted real numbers (not floating-point representations) and wrote out the answer explicitly. It's easy to see that we started with 20 significant figures and ended up with 11 significant figures, even though we're dealing with real numbers/infinite precision. We now turn to the second case, which is the carrying out of this subtraction using floating-point numbers. We can use Python (doubles) to be explicit. We have:

```
>>> 1.2345678912345678912 - 1.2345678900000000000
1.234568003383174e-09
```

Comparing to the answer we had above (for real numbers), we see that we only match the first 6 (out of 11) significant figures. This is partly a result of the fact that each of our initial

[15] Note, however, that ϵ_m was defined in a related but slighty different context.

[16] Incidentally, though we loosely use the term significant figures here and elsewhere, it's better to keep in mind the relative error, instead, which is a more precise and base-independent measure.

numbers is not represented on the computer using the full 20 significant figures, but only at most 16 digits. Explicitly:

```
>>> 1.2345678912345678912
1.234567891234568
>>> 1.2345678900000000000
1.23456789
>>> 1.234567891234568 - 1.23456789
1.234568003383174e-09
```

This shows that we lose precision in the first number, which then has to lead to loss of precision in the result for the subtraction.

It's easy to see that things are even worse than that, though: using real numbers, the subtraction $1.234567891234568 - 1.23456789$ would give us 0.000000001234568. Instead of that, we get `1.234568003383174e-09`. This is not hard to understand: a number like `1.234567891234568` typically has an absolute error in the last digit, i.e., of the order of 10^{-15}, so the result of the subtraction generally cannot be trusted beyond that absolute order (`1.234568003383174e-09` can be rewritten as `1234568.003383174e-15`). This is a result of the fact that in addition or subtraction the *absolute* error in the result comes from adding the absolute errors in the two numbers.

Just in case the conclusion is still not "clicking", let us try to discuss what's going on using relative errors. Here the exact number is $x = 0.0000000012345678912$ while the approximate value is $\tilde{x} = 0.000000001234568003383174$. Since this is one of those situations where we can actually evaluate the error, let us do so. The definition in Eq. (2.12) gives us $\delta x \approx 9.08684 \times 10^{-8}$. Again, observe that we started with two numbers with relative errors of order 10^{-16}, but subtracting them led to a relative error of order roughly 10^{-7}. Another way to look at this result is to say that we have explicitly evaluated the left-hand side of Eq. (2.51), δx. Now, let us evaluate the right-hand side of that equation. Here $a = 1.2345678912345678912$ and $a - b = 0.0000000012345678912$. The right-hand side comes out to be $|a|\, 2\epsilon_m/|a-b| \approx 2.2 \times 10^{-7}$. Thus, we find that the inequality is obeyed (of course), but the actual relative error is a factor of a few smaller than what the error bound would lead us to believe. This isn't too hard to explain: most obviously, the right-hand side of Eq. (2.51) contains a 2, stemming from the assumption that both $\tilde{a}$ and $\tilde{b}$ have a relative error of ϵ_m, but in our case b didn't change when we typed it into the Python interpreter.

You should repeat the above exercise (or something like it) for the cases of addition, multiplication, or division. It's easy to come up with examples where you add two numbers, one of which is poorly constrained, and then you get a large absolute error in the result (but still not as dramatic as in catastrophic cancellation). On the other hand, since in multiplication and division only the relative errors add up, you can convince yourself that since your starting numbers each have a relative error bound of roughly 10^{-16}, the error bound for the result is at worst twice that, which is typically not that bad.

2.3.5 Comparing Floats

Since only machine numbers can be represented exactly (other numbers are rounded to the nearest machine number), we have to be careful when comparing floating-point numbers. We won't go into the question of how operations with floating-point numbers are actually implemented, but some examples may help explain the core issues.

Specifically, you should (almost) never compare two floating point variables $\tilde{x}$ and $\tilde{y}$ for equality: you might have an analytical expectation that the corresponding two real numbers x and y should be the same, but if the values of $\tilde{x}$ and $\tilde{y}$ are arrived at via different routes, their floating-point representations may well be different. A famous example:

```
>>> xt = 0.1 + 0.2
>>> yt = 0.3
>>> xt == yt
False
>>> xt
0.30000000000000004
>>> yt
0.3
```

The solution is to (almost) never compare two floating-point variables for equality: instead, take the absolute value of their difference, and check if that is smaller than some acceptable threshold, e.g., 10^{-10} or 10^{-12}. To apply this to our example above:

```
>>> abs(xt-yt)
5.551115123125783e-17
>>> abs(xt-yt) < 1.e-12
True
```

which behaves as one would expect.[17]

The above recipe (called an *absolute epsilon*) is fine when comparing natural-sized numbers. However, there are situations where it can lead us astray. For example:

```
>>> xt = 12345678912.345
>>> yt = 12345678912.346
>>> abs(xt-yt)
0.0010013580322265625
>>> abs(xt-yt) < 1.e-12
False
```

This makes it look like these two numbers are really different from each other, though

[17] Well, almost: if you were carrying out the subtraction between 0.30000000000000004 and 0.3 using real numbers, you would expect their difference to be 0.00000000000000004. Instead, since they are floats, the answer turns out to be `5.551115123125783e-17`.

it's plain to see that they aren't. The solution is to employ a *relative epsilon*: instead of comparing the two numbers to check whether they match up to a given small number, take into account the magnitude of the numbers themselves, thereby making the comparison relative. To do so, we first introduce a helper function:

```
>>> def findmax(x,y):
...     return max(abs(x),abs(y))
```

This picks the largest magnitude, which we then use to carry out our comparison:

```
>>> xt = 12345678912.345
>>> yt = 12345678912.346
>>> abs(xt-yt)/findmax(xt,yt) < 1.e-12
True
```

Finally, note that there are situations where it's perfectly fine to compare two floats for equality, e.g., `while 1.+small != 1.:`. This specific comparison works: 1.0 is exactly representable in double-precision, so the only scenario where `1.+small` is equal to `1.0` is when the result gets rounded to `1.0`. Of course, this codicil to our rule (you can't compare two floats for equality, except when you can) in practice appears most often when comparing a variable to a literal: there's nothing wrong with saying `if xt == 10.0:` since 10.0 is a machine number and `xt` can plausibly round up or down to that value.

2.4 Rounding Errors in the Wild

Most of what we've had to say about roundoff error up to this point has focused on a single elementary mathematical operation (e.g., one subtraction or one addition). Of course, in actual applications one is faced with many more calculations (e.g., taking the square root, exponentiating), often carried out in sequence. It is often said that rounding error for the case where many iterations are involved leads to roundoff error buildup. This is not incorrect, but more often than not we are faced with one or two iterations that cause a problem, which can typically not be undone after that point. Thus, in the present section we turn to a study of more involved cases of rounding error.

2.4.1 Are Roundoff Errors Random?

At this point, many texts on computational science discuss a standard problem, that of roundoff error propagation when trying to evaluate the sum of n numbers $x_0, x_1, \ldots, x_{n-1}$. One way to go about this is by applying our discussion from section 2.2.2. If each number has the same error bound ϵ, it's easy to convince yourself that since the absolute errors will add up, you will be left with a total error bound that is $n\epsilon$. This is most likely too

pessimistic: it's hard to believe that the errors for n numbers would never cancel, i.e., they would all have the same sign and maximal magnitude.

What's frequently done, instead, is to assume that the errors in the different terms are independent, use the theory of random variables, and derive a result for the scaling with n of the absolute or relative error for the sum $\sum_{i=0}^{n-1} x_i$. We will address essentially the same problem when we introduce Monte Carlo integration in chapter 7. There we will find that, if ϵ is the standard deviation for one number, then the standard error for the sum turns out to be $\sqrt{n}\epsilon$. For large n, it's clear that $\sqrt{n}$ grows much more slowly than n. Of course, this is comparing apples (maximal errors) with oranges (standard errors).

One has to pause, for a minute, however, to think about the assumption that the errors of these $x_0, x_1, \ldots, x_{n-1}$ are stochastically independent in general. As it so happens, many of the examples we will discuss in the rest of this chapter would have been impossible if the errors were independent. To take one case, the next contribution in a Taylor expansion is clearly correlated to the previous contribution. Rounding errors are not random, are often correlated, and usually look like discrete (i.e., not continuous) variables. These are points that, while known to experts (see, e.g., N. Higham's book [39]), are too often obscured in introductory textbooks. The confusion may arise from the fact that the roundoff error in the finite-precision *representation* of a real number may be modelled using a simple distribution; this is wholly different from the roundoff error in a *computation* involving floating-point numbers, which is most certainly not random.

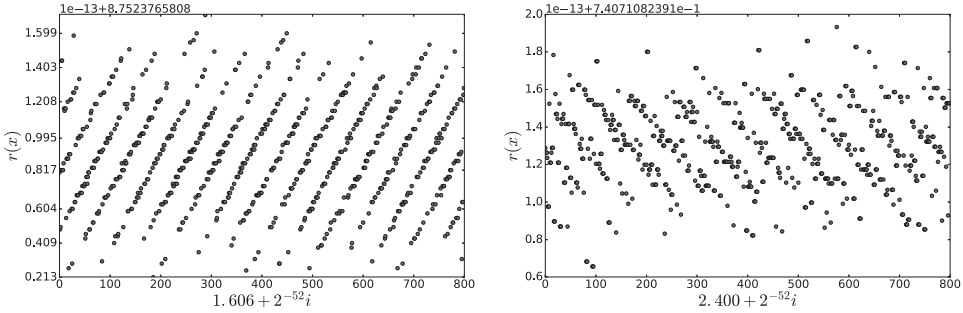
Perhaps it's best to consider a specific example. Let's look at the rational function:

$$r(x) = \frac{4x^4 - 59x^3 + 324x^2 - 751x + 622}{x^4 - 14x^3 + 72x^2 - 151x + 112} \tag{2.53}$$

In the problem set you will learn to code up polynomials using an efficient and accurate approach known as Horner's rule. In the following problem, you will apply what you've learned to this specific rational function. You will find what's shown in Fig. 2.3: this is clear evidence that our method of evaluating the function is sensitive to roundoff error: the typical error is $\approx 10^{-13}$. What's more, the roundoff error follows striking patterns: the left panel shows that the error is not uniformly random.[18] To highlight the fact that the pattern on the left panel is not simply a fluke, the right panel picks a different region and finds a different pattern (again, a mostly non-random one). In the aforementioned problem, you will not only reproduce these results, but also see how one could do better.

Where does this leave us? Since assuming totally independent standard errors is not warranted and assuming maximal errors is too pessimistic, how do we proceed? The answer is that one has to approach each problem separately, so there is no general result for the scaling with n (beyond the formal relation in Eq. (2.44)). As noted, often only a few rounding errors are the dominant contributions to the final error, so the question of finding a scaling with n is moot. We hope that by seeing several cases of things going wrong (and how to fix them), the reader will learn to identify the main classes of potential problems.

[18] In which case we would be seeing "noise" instead of straight lines.



Value of rational function discussed in the main text, at two different regions **Fig. 2.3**

2.4.2 Compensated Summation

We now turn to a crucial issue regarding operations with floats; in short, due to roundoff errors, when you're dealing with floating-point numbers the associative law of algebra does not necessarily hold. You know that `0.1` added to `0.2` does not give `0.3`, but things are even worse than that: *the result of operations involving floating-point numbers may depend on the order in which these operations are carried out*. Here's a simple example:

```
>>> (0.7 + 0.1) + 0.3
1.0999999999999999
>>> 0.7 + (0.1 + 0.3)
1.1
```

It's pretty clear from the above example that, once again, operations that "should" give the same answers (i.e., that *do* give the same answer when dealing with real numbers) may not. This behavior is more than just a curiosity: it can have real-world consequences. In fact, here's an even more dramatic example:

```
>>> xt = 1.e20; yt = -1.e20; zt = 1.
>>> (xt + yt) + zt
1.0
>>> xt + (yt + zt)
0.0
```

In the first case, the two large numbers, `xt` and `yt`, cancel each other and we are left with the unit as the answer. In the second case, we face a situation similar to that in subsection 2.3.3: adding 1 to the (negative) large number `yt` simply rounds to `yt`; this is analogous to the `1. + small` we encountered earlier, only this time we're faced with `big + 1.` and it is the unit that is dropped. Then, `xt` and `yt` cancel each other out (as before). If you're finding these examples a bit disconcerting, you are in good company.

Once you think about the problem more carefully, you might reach the conclusion that the issue that arose here is not too problematic: you were summing up numbers of wildly

Code 2.1 kahansum.py

```
def kahansum(xs):
     s = 0.; e = 0.
     for x in xs:
          temp = s
          y = x + e
          s = temp + y
          e = (temp - s) + y
     return s

if __name__ == '__main__':
     xs = [0.7, 0.1, 0.3]
     print(sum(xs), kahansum(xs))
```

varying magnitudes, so you cannot trust the final answer too much. Unfortunately, as we'll see in section 2.4.4, sometimes you may not even be aware of the fact that the intermediate values in a calculation are large and of opposite signs (leading to cancellations), in which case you might not even know how much you should trust the final answer. A lesson that keeps recurring in this chapter is that you should get used to reasoning about your calculation, in contradistinction to blindly trusting whatever the computer produces.

We don't want to sound too pessimistic, so we will now see how to sum up numbers very accurately. Our task is simply to sum up the elements of a list. In the problem set, we will see that often one can simply sort the numbers and then add them up starting with the smallest one. There are, however, scenarios where sorting the numbers to be summed is not only costly but goes against the task you need to carry out. Most notably, when solving initial-value problems in the study of ordinary differential equations (see chapter 8), the terms *have* to be added in the same order as that in which they are produced.

Here we will employ a nice trick, called *compensated summation* or *Kahan summation*. Qualitatively, what this does is to estimate the rounding error in each addition and then compensate for it with a correction term. More specifically, if you are adding together two numbers (a and b) and $\tilde{s}$ is your best floating-point representation for the sum, then if:

$$e = (a - \tilde{s}) + b \tag{2.54}$$

we can compute $\tilde{e}$ to get an estimate of the error $(a+b)-\tilde{s}$, namely the information that was lost when we evaluated $\tilde{s}$. While this doesn't help us when all we're doing is summing two numbers[19] it can really help when we are summing *many* numbers: add in this correction to the next term in your series, before adding that term to the partial sum.

Typically, compensated summation is more accurate when you are adding a large number of terms, but it can also be applied to the case we encountered at the start of the present

[19] $\tilde{s} + \tilde{e}$ doesn't get you anywhere, since $\tilde{s}$ was already the best we could do!

section. Code 2.1 provides a Python implementation. This does what we described around Eq. (2.54): it estimates the error in the previous addition and then compensates for it. Note how our new function does not need any "length" arguments, since it simply steps through the elements of the list. We then encounter a major new syntactic feature of Python: the line `if __name__ == '__main__':` checks to see if we're running the present file as the main program (which we are). In this case, including this extra check is actually unnecessary: we could have just defined our function and then called it. We will see the importance of this further check later, when we wish to call `kahansum()` without running the rest of the code. The output is:

```
1.0999999999999999 1.1
```

Thus, this simple function turns out to cure the problem we encountered earlier on. We don't want to spend too much time on the topic, but you should play around with compensated summation, trying to find cases where the direct sum does a poor job (here's another example: `xs = [123456789 + 0.01*i for i in range(10)]`).

Even our progress has its limitations: if you take `xs = [1.e20, 1., -1.e20]`, which is (a modified version of) the second example from the start of this section, you will see that compensated summation doesn't lead to improved accuracy. In general, if:

$$\sum_i |x_i| \gg \left| \sum_i x_i \right| \tag{2.55}$$

then compensated summation is not guaranteed to give a small relative error. On a different note, Kahan summation requires more computations than regular summation: this performance penalty won't matter to us in this book, but it may matter in real-world applications.

2.4.3 Naive vs Manipulated Expressions

We now go over a simple example showing how easy it is to lose accuracy if one is not careful; at the same time, we will see how straightforward it is to carry out an analytical manipulation that avoids the problem. The task at hand is to evaluate the function:

$$f(x) = \frac{1}{\sqrt{x^2+1} - x} \tag{2.56}$$

for large values of x. A Python implementation using list comprehensions is given in Code 2.2. The output of running this code is:

```
10000 19999.99977764674
100000 200000.22333140278
1000000 1999984.77112922
10000000 19884107.85185185
```

The answer appears to be getting increasingly worse as the x is increased. Well, maybe: this all depends on what we expect the correct answer to be. On the other hand, the

Code 2.2 naiveval.py

```
from math import sqrt

def naiveval(x):
     return 1/(sqrt(x**2 + 1) - x)

xs = [10**i for i in range(4,8)]
ys = [naiveval(x) for x in xs]
for x, y in zip(xs, ys):
     print(x, y)
```

code/expression we are using is certainly not robust, as you can easily see by running the test case of $x = 10^8$. In Python this leads to a `ZeroDivisionError` since the terms in the denominator are evaluated as being equal. This is happening because for large values of x, we know that $x^2 + 1 \approx x^2$. We need to evaluate the square root very accurately if we want to be able to subtract a nearly equal number from it.

An easy way to avoid this problem consists of rewriting the starting expression:

$$f(x) = \frac{1}{\sqrt{x^2+1} - x} = \frac{\sqrt{x^2+1} + x}{(\sqrt{x^2+1} - x)(\sqrt{x^2+1} + x)} = \frac{\sqrt{x^2+1} + x}{x^2 + 1 - x^2} = \sqrt{x^2+1} + x \tag{2.57}$$

In the second equality we multiplied numerator and denominator with the same expression. In the third equality we used a well-known identity in the denominator. In the fourth equality we cancelled terms in the denominator. Notice that this expression no longer requires a subtraction. If you implement the new expression, you will get the output:

```
10000 20000.000050000002
100000 200000.00000499998
1000000 2000000.0000005001
10000000 20000000.000000052
```

The errors now behave much better: for $x \gg 1$ we have $x^2 + 1 \approx x^2$, so we are essentially printing out $2x$. There are several other cases where a simple rewriting of the initial expression can avoid bad numerical accuracy issues (often by avoiding a subtraction).

2.4.4 Computing the Exponential Function

We now turn to an example where several calculations are carried out in sequence. Thus, if something goes wrong we must carefully sift through intermediate results to see what went wrong (and when). We focus on the task of computing the exponential function (assuming

we have no access to a math library), by using the Taylor/Maclaurin series:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots \tag{2.58}$$

We're clearly not going to sum infinitely many terms, so we approximate this expansion by keeping only the terms up to n_{max}:

$$e^x \approx \sum_{n=0}^{n_{max}} \frac{x^n}{n!} \tag{2.59}$$

A naive implementation of this algorithm would divide x raised to increasingly large powers with increasingly large factorials, summing the result of such divisions up to a specified point. This approach suffers from (at least) two problems: (a) calculating the power and the factorial is costly,[20] and (b) both x^n and $n!$ can become very large numbers (potentially overflowing) even though their ratio can be quite small.

Instead of calculating the power and factorial (only to divide them away), we take advantage of the fact that the n-th term in the expansion can be related to the $(n-1)$-th term:

$$\frac{x^n}{n!} = \frac{x}{n}\frac{x^{n-1}}{(n-1)!} \tag{2.60}$$

Thus, we can get a new term by multiplying the old term by x/n. This leads to a straightforward implementation that obviates the calculation of powers and factorials. Incidentally, it is easy to see that the magnitude of the terms grows, if $x > n$ holds; we will later examine the consequences of this fact.

Before turning to an implementation in Python, let us think about when to terminate our summation loop. There are generally two possibilities: (a) either we test for when the new term is "small", or (b) we test for when the running total has reached a desirable value. At first sight, it is difficult to implement (b), since we don't know the correct answer for the sum, so we turn to (a): this in its turn can be accomplished in (at least) two ways. First, we could terminate when the n-th term is a small fraction of the running total (say, less than 10^{-8}). This, however, seems needlessly restrictive, bringing us to: second, we could simply terminate when the n-th term underflows to zero. A moment's reflection, however, brings us back to point (b): at the end of the calculation, we're not really interested in the n-th term, but in the total sum. Thus, a better approach is to terminate the loop when it is determined that adding the n-th term to the running total doesn't change the sum.[21]

The above ideas are straightforwardly implemented in Python in Code 2.3. Clearly, the test of terminating the loop when the latest term doesn't change the answer takes the form `while newsum != oldsum:` and the production of the latest term using Eq. (2.60) is given by `term *= x/n`. We also print a counter, the running sum, and the latest term; you can comment this line out later on. The loop in the main program calls our function for three different values of x, which we now discuss one at a time. The output for $x = 0.1$ is:

[20] Not to mention that the latter would have to be coded up separately, if we're not using a math library.

[21] Take some time to understand this: we care about whether the latest term changes the answer or not, regardless of whether or not the last term on its own underflows to zero.

Code 2.3 compexp.py

```
from math import exp

def compexp(x):
     n = 0
     oldsum, newsum, term = 0., 1., 1.
     while newsum != oldsum:
          oldsum = newsum
          n += 1
          term *= x/n
          newsum += term
          print(n, newsum, term)
     return newsum

for x in (0.1, 20., -20.):
     print("x, library exp(x):", x, exp(x))
     val = compexp(x)
```

```
x, library exp(x): 0.1 1.1051709180756477
1 1.1 0.1
2 1.105 0.005000000000000001
3 1.1051666666666666 0.0001666666666666667
...
8 1.1051709180756446 2.480158730158731e-13
9 1.1051709180756473 2.75573192239859e-15
10 1.1051709180756473 2.75573192239859e-17
```

where we suppressed part of the output (as we will continue to do below). Note that if we limit ourselves to $n_{max} = 2$ then the answer we get is `1.105`. Using the language introduced around Eq. (2.4), this result suffers only from *approximation error*, not from *roundoff error* (and obviously not from any roundoff error buildup): even if we were using real numbers (of infinite precision) to do the calculation for $n_{max} = 2$, we would have found 1.105.

In this case, since x is small, we observe that the value of `term` is decreasing with each new iteration. As advertised, the loop terminates when the value of `term` is small enough that it doesn't change the value of `newsum`. Comparing our final answer for $e^{0.1}$ with that provided by the `math` module's `exp()` function, we find agreement in 16 decimal digits, which is all one can hope for when dealing with double-precision floating-point numbers. We note, finally, that the convergence was achieved after only 10 steps.

We next turn to the output for $x = 20$:

```
x, library exp(x): 20.0 485165195.4097903
1 21.0 20.0
2 221.0 200.0
3 1554.3333333333335 1333.3333333333335
...
18 185052654.63711208 40944813.9157307
19 228152458.75893387 43099804.12182178
20 271252262.88075566 43099804.12182178
21 312299695.3777288 41047432.49697313
...
66 485165195.4097904 1.3555187344975148e-07
67 485165195.40979046 4.046324580589596e-08
68 485165195.40979046 1.1900954648792928e-08
```

We immediately observe that this case required considerably more iterations to reach convergence: here the final value of the sum is $\approx 5 \times 10^8$ and the smallest `term` is $\approx 1 \times 10^{-8}$.[22] In both cases, there are 16 or 17 orders of magnitude separating the final answer for the sum from the smallest `term`. Observe that the magnitude of `term` here first increases until a maximum, at which point it starts decreasing. We know from our discussion of Eq. (2.60) that the terms grow as long as $x > n$; since $x = 20$ here, we see that $n = 20$ is the turning point after which the terms start decreasing in magnitude.

Comparing our final answer for e^{20} with that provided by the `math` module's `exp()` function, we find agreement in 15 decimal digits. This is certainly not disappointing, given that we are dealing with doubles. We observe, for now, that the error in the final answer stems from the last digit in the 20th term; we will further elucidate this statement below.

Up to this point, we've seen very good agreement between the library function and our numerical sum of the Taylor expansion, for both small and large values of x. We now turn to the case of negative x. The output for $x = -20$ is:

```
x, library exp(x): -20.0 2.061153622438558e-09
1 -19.0 -20.0
2 181.0 200.0
3 -1152.3333333333335 -1333.3333333333335
...
18 21277210.34254431 40944813.9157307
19 -21822593.779277474 -43099804.12182178
20 21277210.34254431 43099804.12182178
21 -19770222.154428817 -41047432.49697313
...
93 6.147561828914624e-09 -8.56133790667976e-24
94 6.147561828914626e-09 1.8215612567403748e-24
```

[22] Compare with the case of $x = 0.1$, where the sum was of order 1 and the smallest `term` was $\approx 2 \times 10^{-17}$.

```
95 6.147561828914626e-09 -3.8348658036639467e-25
```

In this case, too, we observe that the code required considerably more iterations to reach convergence (even more than what was needed for $x = 20$). The final answer for the sum here is much smaller than before ($\approx 6 \times 10^{-9}$), so we need to wait until `term` becomes roughly 16 orders of magnitude smaller than that ($\approx 4 \times 10^{-25}$). Again, observe that the magnitude of `term` here first increases until a maximum, at which point it starts decreasing. Just like in the previous case, the magnitude of `term` stops increasing after $n = 20$. As a matter of fact, the absolute value of every single `term` here is the same as it was for $x = 20$, as can be seen here by comparing the output for lines 1–3 and 18–21 to our earlier output.

Comparing our final answer for e^{-20} with that provided by the `math` module's `exp()` function, we find (the right order of magnitude, but) absolutely no decimal digits agreeing! In other words, for $x = -20$ our sum of the Taylor series is totally wrong.

Let us try to figure out what went wrong. We observe that the magnitude of `newsum` in the first several iterations is clearly smaller than the magnitude of `term`: this means that in addition to the absolute value of `term` growing, it's growing faster than the absolute value of `newsum`. This difference in the speed of growth becomes more dramatic in the 20th iteration, where `term` is more than two times bigger than the absolute value of `newsum`. What is the difference between the present case and that of $x = 20$? Clearly, it's related to the fact that the sign of `term` here oscillates from iteration to iteration. This is a result of the fact that $(-20)^n = -20^n$ for n odd (take Eq. (2.59) and set $x = -20$). Since the `term`s have alternating signs, they cancel each other and at some point `newsum` (which itself also oscillated in sign for a while) starts to get smaller and smaller.

We've now seen why the negative x case is different: there is cancellation between numbers of comparable magnitude. We can do even better than this handwaving explanation, though. Each `term` is accurate to at most 16 decimal digits (since it's a double-precision floating-point number). Thus, the largest-magnitude `term` has an absolute error in its last digit, of order 10^{-8}. (Another way to say this is that every double has a relative error of roughly 10^{-16} and since this specific double has magnitude 10^8 it has an absolute error of roughly 10^{-8}.) Note that we are not here talking about the smallest `term` (adding which leaves `newsum` unchanged) but about the *largest* `term` (which has the largest error of all terms): as we keep adding more terms to `newsum`, this largest error is not reduced but is actually propagated over! Actually, things are even worse than that: the final answer for `newsum` has magnitude $\approx 6 \times 10^{-9}$ and is therefore even smaller than the error of $\approx 10^{-8}$ that we've been carrying along. Thus, the final answer has no correct significant digits! In the case of positive x (namely the $x = 20$ we studied above), the largest `term` also had an error of $\approx 10^{-8}$, but since in that case there were no cancellations, the final value of the sum was $\approx 5 \times 10^8$, leading to no error for the first 15 decimal digits.

We thus see that our algorithm for calculating e^x for negative x (via the Taylor expansion) is unstable because it introduces cancellation. As a matter of fact, in this specific case the cancellation was needless: we could have just as easily taken advantage of $e^x = 1/e^{-x}$ and then carried out the sum for a positive value of x (which would not have suffered from cancellation issues) and then proceeded to invert the answer at the end. In our example, we can estimate e^{-20} by summing the Taylor series for e^{20} and then inverting the answer:

```
>>> 1/485165195.40979046
2.061153622438557e-09
```

We have 15 digits of agreement! This shouldn't come as a surprise: the Taylor expansion for positive x doesn't contain any cancellations and leads to 15–16 correct significant digits. After that, dividing 1 with a large number with 15–16 correct significant digits leads to an answer with 15–16 correct significant digits. Of course, it's obvious that this quick fix does not necessarily apply to all Taylor expansions: some of them have the alternating signs built-in.[23] Those cases, too, however, can typically benefit from analytical manipulations that make the problem better-behaved numerically.

You should keep in mind that there are situations where the result blows up in unexpected ways, despite the fact that we weren't carrying out any subtractions. We revisit this in the problems at the end of the chapter.

2.4.5 An Even Worse Case: Recursion

In the previous subsection we examined a case where adding together several correlated numbers leads to an error (in the largest one) being propagated unchanged to the final answer, thereby making it inaccurate. In this subsection, we will examine the problem of recursion, where even a tiny error in the starting expression can be multiplied by the factorial of a large number, thereby causing major headaches.

Our task is to evaluate integrals of the form:

$$f(n) = \int_0^1 x^n e^{-x} dx \tag{2.61}$$

for different (integer) values of n, i.e., for $n = 0, 1, 2, \ldots$. To see how this becomes a recursive problem, we analytically evaluate the indefinite integral for the first few values of n:

$$\begin{aligned} &\int e^{-x} dx = -e^{-x}, && \int x e^{-x} dx = -e^{-x}(x+1), \\ &\int x^2 e^{-x} dx = -e^{-x}(x^2+2x+2), && \int x^3 e^{-x} dx = -e^{-x}(x^3+3x^2+6x+6) \end{aligned} \tag{2.62}$$

where we used integration by parts. This leads us to a way of relating the indefinite integral for the n-th power to the indefinite integral for the $(n-1)$-th power:

$$\int x^n e^{-x} dx = n \int x^{n-1} e^{-x} dx - x^n e^{-x} \tag{2.63}$$

It is now trivial to use this result in order to arrive at a recursive expression for the definite integral $f(n)$ in Eq. (2.61):

$$f(n) = n f(n-1) - e^{-1} \tag{2.64}$$

This works for $n = 1, 2, 3, \ldots$ and we already have the result $f(0) = 1 - e^{-1}$.

[23] For example, that for $\sin x$ or $\cos x$, which you will encounter in a problem.

Code 2.4 recforw.py

```
from math import exp

def forward(nmax=22):
     oldint = 1 - exp(-1)
     for n in range(1,nmax):
          print(n-1, oldint)
          newint = n*oldint - exp(-1)
          oldint = newint

print("n = 20 answer is 0.0183504676972562")
print("n, f[n]")
forward()
```

We code this up in the most straightforward way possible in Code 2.4.[24] This clearly shows that we only need to keep track of two numbers at any point in time: the previous one and the current one. We start at the known result $f(0) = 1 - e^{-1}$ and then simply step through Eq. (2.64); note that our function is simply printing things out, not returning a final value. We start at $n = 0$ and the last value we print out is for $n = 20$; we're also providing the correct answer, arrived at via other means. We immediately notice the difference between the algorithm coded up in the previous subsection (where we were simply adding in an extra number) and what's going on here: even if we ignore any possible subtractive cancellation, Eq. (2.64) contains $nf(n-1)$ which means that any error in determining $f(n-1)$ is *multiplied* by n when producing the next number. If n is large, that can be a big problem. Since the expression we're dealing with is recursive, even if we start with a tiny error, this is compounded by being multiplied by n every time through the loop. We get:

```
n = 20 answer is 0.0183504676972562
n, f[n]
0 0.6321205588285577
1 0.26424111765711533
2 0.16060279414278833
...
16 0.022201910404060943
17 0.009553035697593693
18 -0.19592479861475587
19 -4.090450614851804
20 -82.17689173820752
```

[24] Note that we chose not to employ a recursive Python function to implement Eq. (2.64).

recback.py Code 2.5

```
from math import exp

def backward(nmax=31):
     oldint = 0.01
     for n in reversed(range(20,nmax)):
          print(n, oldint)
          newint = (oldint + exp(-1))/n
          oldint = newint

print("n = 20 answer is 0.0183504676972562")
print("n, f[n]")
backward()
```

Clearly, that escalated fast. Even though for the first 15 or so terms we see a gradual decline in magnitude, after that the pace picks up pretty fast. We are dealing with a numerical instability, which ends up giving us garbage for $f(20)$. It's not hard to see why: since $f(0)$ is stored as a double-precision floating-point number, it has an absolute error in its last digit, of order 10^{-16}. Focusing only on the $nf(n-1)$ term in Eq. (2.64), we see that by the time we get up to $n = 20$, our 10^{-16} error will have been multiplied by 20!: given that $20! \times 10^{-16} \approx 243$, this completely overwhelms our expected answer of ≈ 0.018.

The process we followed in the code above, starting at $n = 0$ and building up to a finite n, is called *forward recursion*. We will eliminate our headaches by a simple trick, namely the use of *backward recursion*: solve Eq. (2.64) for $f(n-1)$ in terms of $f(n)$:

$$f(n-1) = \frac{f(n) + e^{-1}}{n} \tag{2.65}$$

The way to implement this new equation is to start at some large value of n, say $n = 30$, and then step *down* one integer at a time. We immediately realize that we don't actually know the value of $f(30)$. However, the algorithm implied by Eq. (2.65) is much better behaved than what we were dealing with before: even if we start with a bad estimate of $f(30)$, say with an error of 0.01, the error will be *reduced* with each iteration, since we are now *dividing* with n. Thus, by the time we get down to $n = 20$, the error will have turned into $0.01/(30 \times 29 \times 28 \times \cdots \times 22 \times 21) \approx 9 \times 10^{-17} \approx 10^{-16}$, which happens to be quite good. Code 2.5 shows an implementation of backward recursion; running this, we get:

```
n = 20 answer is 0.0183504676972562
n, f[n]
30 0.01
29 0.012595981372381411
28 0.013119842156683579
```

```
...
22 0.016688929189184395
21 0.01748038047093758
20 0.018350467697256186
```

We have agreement in the first 14 significant figures, which means that the fifteenth significant figure is off. This is a result of the aforementioned absolute error of $\approx 10^{-16}$; notice how the first significant figure is of order 10^{-2}.

2.4.6 When Rounding Errors Cancel

We now turn to a slightly more complicated example, which shows how function evaluations can be rather counterintuitive. The moral to be drawn from our discussion is that there is no substitute for thinking. As a matter of fact, this is a case where an approximation that seems to be bad at first sight ends up performing much better than we had expected.

Our goal is to examine the behavior of the function $f(x) = (e^x - 1)/x$ at small (or perhaps intermediate) x. We start from coding this up in Python in the obvious (naive) way, see the function `f()` in Code 2.6. The output of running this code is:

```
1e-14 0.9992007221626409 1.0000000000000005
1e-15 1.1102230246251565 1.0000000000000004
1e-16 0.0 1.0
-1e-15 0.9992007221626408 0.9999999999999994
-1e-16 1.1102230246251565 1.0
-1e-17 -0.0 1.0
```

Ignore the function `g()` and the last number in each row, for now. We see that for small x (whether negative or positive) the naive function gives not-very-accurate results, until at some point for even smaller x the answer is absolutely wrong.[25] It's obvious that our code is plagued by catastrophic cancellation.

One way to go about improving the solution would be to use the Maclaurin series for e^x:

$$\frac{e^x - 1}{x} \approx 1 + \frac{x}{2!} + \frac{x^2}{3!} + \cdots \tag{2.66}$$

The problem with this is that to get a desired accuracy we need to keep many terms (and the number of terms depends on the desired accuracy). The trick in the function `g()`, instead, nicely makes use of standard rounding properties: it compares a float to a float-literal for equality. This is perfectly fine, as the relevant lines of code are there precisely to catch the cases where `w` is rounded to 1 or to 0. In the former case (e^x rounding to 1), it returns the analytical answer by construction. In the latter case (e^x rounding to 0, when x is large and negative), it plugs in the value for the rest of the expression. That leaves us with the crucial expression `(w-1)/log(w)`, which is equivalent to `(exp(x)-1)/log(exp(x))`, for all other cases.

[25] In the limit $x \to 0$ we know the answer must be 1, from L'Hôpital's rule.

cancel.py Code 2.6

```
from math import exp, log

def f(x):
     return (exp(x) - 1)/x

def g(x):
     w = exp(x)
     if w==0.:
          val = -1/x
     elif w==1.:
          val = 1.
     else:
          val = (w-1)/log(w)
     return val

xs = [10**(-i) for i in (14, 15, 16)]
xs += [-10**(-i) for i in (15, 16, 17)]
fvals = [f(x) for x in xs]
gvals = [g(x) for x in xs]
for x, fval, gval in zip(xs, fvals, gvals):
     print(x, fval, gval)
```

Let us now look at the last number printed on each row, which corresponds to `g()`. It's obvious that the revised version is much better than the naive one. The reason this new function works so well is because it makes the exponential appear in both the numerator and the denominator: `(exp(x)-1)/log(exp(x))`. As a result, the roundoff error in the evaluation of the exponential `exp(x)` in the numerator is cancelled by the presence of the same roundoff error in the exponential `exp(x)` in the denominator.

This result is sufficiently striking that it should be repeated: our updated function works better because it plays off the error in `exp(x)` in the numerator, against the same error in `exp(x)` in the denominator. Let's study the case of $x = 9 \times 10^{-16}$ in detail. The algorithm in `f()` does the following calculation:

$$\frac{\mathtt{e}^{\mathtt{x}} - 1}{\mathtt{x}} = \frac{8.881784197001252\mathtt{e} - 16}{9\mathtt{e} - 16} = 0.9868649107779169 \tag{2.67}$$

where we are also showing the intermediate results that Python produces. Similarly, the algorithm in `g()` does the following calculation:

$$\frac{\mathtt{e}^{\mathtt{x}} - 1}{\log(\mathtt{e}^{\mathtt{x}})} = \frac{8.881784197001252\mathtt{e} - 16}{8.881784197001248\mathtt{e} - 16} = 1.0000000000000004 \tag{2.68}$$

Given the earlier outputs, you should not be suprised that the final answer is more accurate. But look at the same calculation as that in `g()`, this time carried out using real numbers:

$$\frac{e^x - 1}{\log(e^x)} = \frac{9.000000000000004050\ldots \times 10^{-16}}{9.000000000000000000\ldots \times 10^{-16}} = 1.000000000000000450\ldots \tag{2.69}$$

The algorithm in `g()`, when employing floating-point numbers, produces an inaccurate numerator (`e^x - 1`) and an inaccurate denominator (`log(e^x)`) that are divided to produce an accurate ratio. There is a lesson to learn here: we usually only look at intermediate results when something goes wrong. In the case under study, we found that the intermediate results are actually bad, but end up leading to desired behavior in the end.

The reasonably simple example we've been discussing here provides us with the opportunity to note a general lesson regarding numerical accuracy: it is quite common that mathematically elegant expressions, like `f()`, are numerically unreliable. On the other hand, numerically more accurate approaches, like `g()`, are often messier, containing several (ugly) cases. There's often no way around that.

2.5 Project: the Multipole Expansion in Electromagnetism

Our physics project involves a configuration of several point charges and the resulting electrostatic potential; we will then introduce what is known as the *multipole expansion*, whereby the complications in a problem are abstracted away and a handful of numbers are sufficient to give a good approximation to physical properties. We will be applying this to the Coulomb potential but, unsurprisingly, the same approach has also been very fruitful in the context of the Newton potential, i.e., the gravitational field of planets and stars.

Since this chapter has focused on numerics, so will our project: we will encounter series convergence as well as recurrence relations. We use as our starting point the electric-field-visualization machinery that we introduced in section 1.7. In our discussion of the multipole expansion we will introduce what are known as *Legendre polynomials*. In addition to helping us simplify the study of electrostatics in what follows, these will also make appearances in later chapters.

2.5.1 Potential of a Distribution of Point Charges

General Case

In the project at the end of the previous chapter we focused on the electric field, which is a vector quantity. In most of the present section, we will, instead, be studying the electrostatic potential. As you may recall from a course on electromagnetism, since the curl of the electric field $\mathbf{E}$ is zero, there exists a scalar function whose gradient is the electric field:

$$\mathbf{E}(\mathbf{r}) = -\nabla\phi(\mathbf{r}) \tag{2.70}$$

where $\phi(\mathbf{r})$ is called the *electrostatic potential*. In general there would also be a second term on the right-hand side, involving the vector potential $\mathbf{A}$, but here we are focusing only on the problem of static point charges.

The electrostatic potential at point P (located at $\mathbf{r}$) due to the point charge q_0 (which is located at $\mathbf{r}_0$) is simply:

$$\phi_0(\mathbf{r}) = k\frac{q_0}{|\mathbf{r}-\mathbf{r}_0|} \tag{2.71}$$

where, as before, Coulomb's constant is $k = 1/(4\pi\epsilon_0)$ in SI units (and ϵ_0 is the permittivity of free space). As you can verify by direct differentiation, this leads to the electric field in Eq. (1.2).[26] If we were faced with more than one point charge, we could apply the *principle of superposition* also to the electrostatic potential, similarly to what we did for the electric-field vector in Eq. (1.3). As a result, when dealing with the n point charges $q_0, q_1, \ldots, q_{n-1}$ located at $\mathbf{r}_0, \mathbf{r}_1, \ldots, \mathbf{r}_{n-1}$ (respectively), namely the configuration shown in Fig. 1.5, the total electrostatic potential at the location $\mathbf{r}$ is:

$$\phi(\mathbf{r}) = \sum_{i=0}^{n-1} \phi_i(\mathbf{r}) = \sum_{i=0}^{n-1} k\frac{q_i}{|\mathbf{r}-\mathbf{r}_i|} \tag{2.72}$$

i.e., a sum of the individual potential contributions. Obviously, this $\phi(\mathbf{r})$ is a scalar quantity. It contains $|\mathbf{r}-\mathbf{r}_i|$ in the denominator: this is the celebrated *Coulomb potential*.

Example: Array of 36 Charges

For the sake of concreteness, throughout this project we will study an arbitrarily chosen specific configuration: 36 charges placed in a square from −0.5 to 0.5 in both x and y directions. To keep things interesting, we will pick the q_i charges to have varying magnitudes and alternating signs. Using our code `vectorfield.py` from the previous chapter we are led to Fig. 2.4 for this case, where this time we are using a color map (in grayscale) instead of line width to denote the field strength.[27] This array of charges, distributed across six rows and six columns, is here accompanied by the electric field, similarly to what we saw in Fig. 1.6. In the present section we are interested in the electrostatic potential, not directly in the electric field, but you should be able to go from the former to the latter using Eq. (2.70). Our new figure shows many interesting features: looking at the left panel, the direction and magnitude of the electric field are quite complicated; the field appears to be strongest in between the charges. As we move farther away from the charge array, however, as shown in the right panel, we start noticing the big picture: the field is strongest inside the array, then we notice some intermediate-scale features at negative y, and then as we get farther from the configuration of charges the electric field appears reasonably simple. We recall that positive charges act as "sources": since at large distances all the arrows point outward, we see that, abstracting away all the details, "effectively" the charge array acts

[26] We could have included an arbitrary offset here, but we haven't.
[27] Here and below x and y are measured in meters.

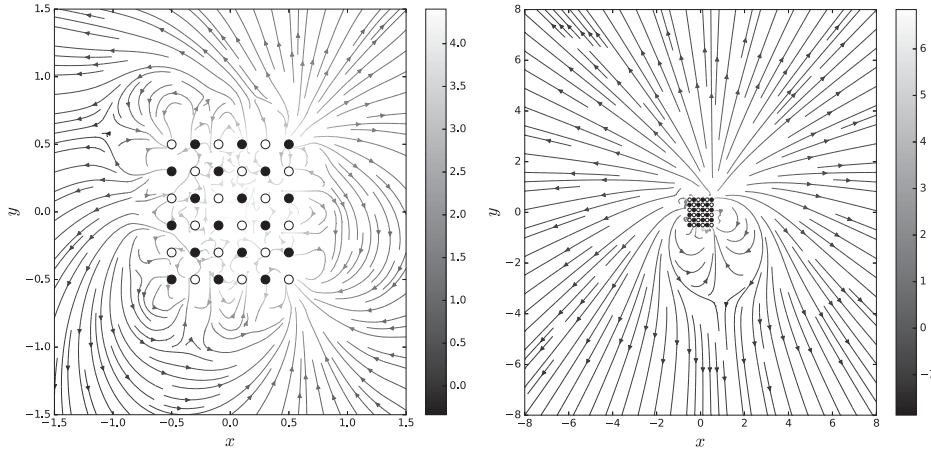
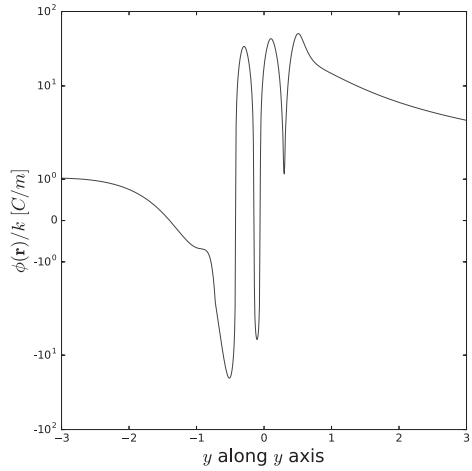


Fig. 2.4 Array of 36 charges of alternating sign and varying magnitudes

like an overall positive charge (though there may be more complicated residual effects not captured by this simple analogy).

As advertised, our main concern here is the electrostatic potential. The question then arises of how to visualize the potential: in principle, one could draw equipotential surfaces (curves), which would be orthogonal to the electric field lines of Fig. 2.4. Instead, with a view to what we will be studying in the following sections, here we opt for something simpler: we pick a specific direction on the $x-y$ plane and study the potential along it. Since the right panel in Fig. 2.4 exhibits interesting behavior at negative y that is not present for positive y, we decide to study the y axis itself (i.e., set $x = 0$) for both positive and negative y. The result of evaluating $\phi(\mathbf{r})$ along the y axis as per Eq. (2.72) (for the given configuration of charges) is shown in Fig. 2.5, using a symmetrical-log scale.[28] The overall features are easy to grasp: we observe rapid oscillations at short distances, which is where the q_i charges are physically located, and then simpler behavior at larger distances. As we expected based on our electric-field visualization, the potential exhibits more structure (even changing sign) at intermediate distances along the negative y axis than it does along the positive y axis. Speaking of the sign, let's do a quick consistency check: at large distances along the positive y axis we see that the potential gets smaller as we increase y; since $\mathbf{E}(\mathbf{r}) = -\nabla\phi(\mathbf{r})$ we expect a positive electric field (pointing up). Similarly, along the negative y axis the potential gets larger as we increase the y magnitude at intermediate distances, so we expect the electric field, again, to be positive (i.e., point up). As hinted at in the leftmost part of our curve, as you keep increasing the y magnitude along the negative y axis the situation will actually change: at some point for $y \lesssim -3$ you will "curve down", implying a negative electric field (pointing down). All these findings are consistent with what is shown in Fig. 2.4: take a minute to inspect that figure more closely.

[28] Note that we've divided out k, so our values are given in units of C/m.



Electrostatic potential for the 36-charge array, along the y axis **Fig. 2.5**

Implementation

At this point, it's probably best to get a bit more specific: the results in these two figures correspond to a given choice (not only of charge placement and sign but also) of the magnitudes of the charges involved. Thinking about how to code up our 36-charge array, we could go about this task in several different ways, e.g., using distinct lists for charges, x coordinates, and y coordinates. However, since we already employed a Python dictionary in `vectorfield.py` in order to associate charges with their positions, it stands to reason that we should now generalize this approach: instead of filling in the keys and values by hand, we should do so programmatically. Code 2.7 provides a Python implementation. Overall, we see that this code produces the point-charge distribution, introduces a helper function, calculates $\phi(\mathbf{r})$ as per Eq. (2.72), and prints out the potential value at two points along the y axis. It should be easy to see that using these functions you can produce Fig. 2.5. Let's go over each aspect of this program in more detail.

The function `chargearray()` produces the dictionary `qtopos`. You may recall that this was accomplished in one line in the code implementing the project of the previous chapter: there, we were faced with only two (or at most four) charges, so we "hard-coded" the keys and values. To do things in a more systematic way, we first remember how to populate dictionaries: in section 1.3.3 we saw that the syntax is `htow[key] = value`. Producing a grid of 36 elements (6×6) is not too difficult: we could simply pick some values (as in `vals`) and iterate over x and y, adding in a new key/value pair each time (with the keys being the q_i and the values being the $\mathbf{r}_i$). Trying to be idiomatic, we make use of `enumerate()` so we can have access both to the value of the coordinate and to its index.[29]

Our task is complicated somewhat by the requirement that the charges be alternating: if on a given row the first charge is negative, the one to its right should be positive, then the

[29] We even kept things general and did not hard-code the value 6, instead passing it in as an argument.

Code 2.7 chargearray.py

```
from kahansum import kahansum
from math import sqrt

def chargearray(nvals):
     vals = [-0.5 + i/(nvals-1) for i in range(nvals)]
     qtopos = {}
     for i,posx in enumerate(vals):
          for j,posy in enumerate(vals):
               count = j + nvals*i + 1
               key = 1.02*count if (i+j)%2==0 else -count
               qtopos[key] = posx, posy
     return qtopos

def vecmag(rs):
     sq = [r**2 for r in rs]
     return sqrt(kahansum(sq))

def fullpot(qtopos,rs):
     potvals = []
     for q,pos in qtopos.items():
          diffs = [r - po for r,po in zip(rs,pos)]
          R = vecmag(diffs)
          potvals.append(q/R)
     return kahansum(potvals)

if __name__ == '__main__':
     qtopos = chargearray(6)
     for y in 1,-1:
          rs = [0.,y]
          potval = fullpot(qtopos,rs)
          print(rs, potval)
```

next one negative and so on. This isn't too hard to implement, either: you could simply have an index and check whether or not that is even or odd. Unfortunately, in our case we want this to go on as you move to the next row: the last charge on the first row is positive, but the one immediately under it should be negative. In other words, we need to traverse our grid from left to right, then from right to left, and so on, just like in ancient Greek inscriptions.[30]

[30] This pattern is called *boustrophedon*, namely as an ox turns with the plough at the end of the furrow.

This is accomplished in our test `if (i+j)%2==0`. To keep things interesting, we picked each q_i according to when we encountered it, via `count = j + nvals*i + 1`.[31] When constructing `key`, we artificially inflated the positive charge values: as you will discover when you play around with this code, the total charge (adding together all q_i) would have been zero had we not taken this step.

We then introduce an auxiliary function, which evaluates the magnitude of a vector using a list comprehension: observe how convenient Python's `for` is here, since we don't need to specify how many dimensions our vector has. We also employ our very own `kahansum()` from earlier in this chapter: while this is probably overkill for summing just a few numbers, it's a nice opportunity to employ functionality we've already developed. We are, finally, also in a position to see why we had included the line saying `if __name__ == '__main__':` all the way back in `kahansum.py`: since we're now not running that file as the main program, the lines that followed the `if` check are not executed this time. If we hadn't taken care of this check, the older file's output would be confusingly printed out every time we ran the present `chargearray.py` file.

The final function of this program implements Eq. (2.72): we sum the $\phi_i(\mathbf{r})$ contributions one by one. Using the `items()` method of dictionaries really pays off: once again, we don't have to specify ahead of time how many charges we are faced with *or* how many dimensions our vectors have. We simply have access to one q_i and one $\mathbf{r}_i$ at a time. We first form the components of $\mathbf{r} - \mathbf{r}_i$ using a list comprehension as well as Python's `zip()`. We then store each $\phi_i(\mathbf{r})$ contribution and at the end use `kahansum()` to sum them all together. The main program simply picks two arbitrary points on the y axis and prints out the value of the potential $\phi(\mathbf{r})$ at those points; you can use these values as benchmarks later in this section, after we've developed the multipole expansion. Both the values and the signs are consistent (nay, identical) with what we saw in Fig. 2.5.

2.5.2 Expansion for One Point Charge

We now turn to our stated aim, which is to approximate a complicated electrostatic potential using only a few (or several) terms. To keep things manageable, we start from the simple case of a single point charge. This allows us to introduce Legendre polynomials without getting lost in a sea of indices; in the following subsection, we will apply what we've learned to the general case of a distribution of point charges.

First Few Terms

For one point charge q_0 (located at $\mathbf{r}_0$), the electrostatic potential at point P (at position $\mathbf{r}$) is given by Eq. (2.71):

$$\phi_0(\mathbf{r}) = k\frac{q_0}{|\mathbf{r} - \mathbf{r}_0|} \tag{2.73}$$

Roughly speaking, what we'll do is to massage the denominator such that it ends up containing only $\mathbf{r}$ (or $\mathbf{r}_0$) but not the difference between the two vectors.

[31] We add in a unit to make sure we don't include a charge of zero magnitude.

With this in mind, let us examine the square of the denominator:

$$(\mathbf{r}-\mathbf{r}_0)^2 = r^2 + r_0^2 - 2rr_0\cos\theta_0 = r^2\left[1+\left(\frac{r_0}{r}\right)^2 - 2\left(\frac{r_0}{r}\right)\cos\theta_0\right] \equiv r^2\left[1+\alpha\right] \tag{2.74}$$

You may think of the first equality as the so-called law of cosines, or as simply expanding the square and expressing the dot product of the two vectors in terms of the angle θ_0 between them. In the second equality, we pulled out a factor of r^2. In the third equality we noticed the overall structure of the expression, by defining:

$$\alpha \equiv \left(\frac{r_0}{r}\right)\left(\frac{r_0}{r} - 2\cos\theta_0\right) \tag{2.75}$$

We can now use the binomial theorem, a standard Taylor-expansion result:

$$\frac{1}{(1+x)^m} = 1 - mx + \frac{m(m+1)}{2!}x^2 - \frac{m(m+1)(m+2)}{3!}x^3 + \cdots \tag{2.76}$$

We are in a position to expand the Coulomb potential from Eq. (2.73) as follows:

$$\begin{aligned}
\frac{1}{|\mathbf{r}-\mathbf{r}_0|} &= \frac{1}{r}\frac{1}{(1+\alpha)^{1/2}} = \frac{1}{r}\left(1 - \frac{1}{2}\alpha + \frac{3}{8}\alpha^2 - \frac{5}{16}\alpha^3 + \frac{35}{128}\alpha^4 - \cdots\right) \\
&= \frac{1}{r}\Bigg[1 - \frac{1}{2}\left(\frac{r_0}{r}\right)\left(\frac{r_0}{r} - 2\cos\theta_0\right) + \frac{3}{8}\left(\frac{r_0}{r}\right)^2\left(\frac{r_0}{r} - 2\cos\theta_0\right)^2 \\
&\qquad - \frac{5}{16}\left(\frac{r_0}{r}\right)^3\left(\frac{r_0}{r} - 2\cos\theta_0\right)^3 + \frac{35}{128}\left(\frac{r_0}{r}\right)^4\left(\frac{r_0}{r} - 2\cos\theta_0\right)^4 - \cdots\Bigg] \\
&= \frac{1}{r}\Bigg[1 + \left(\frac{r_0}{r}\right)\cos\theta_0 + \left(\frac{r_0}{r}\right)^2\frac{1}{2}\left(3\cos^2\theta_0 - 1\right) + \left(\frac{r_0}{r}\right)^3\frac{1}{2}\left(5\cos^3\theta_0 - 3\cos\theta_0\right) \\
&\qquad + \left(\frac{r_0}{r}\right)^4\frac{1}{8}\left(35\cos^4\theta_0 - 30\cos^2\theta_0 + 3\right) + \cdots\Bigg]
\end{aligned} \tag{2.77}$$

In the first equality we took one over the square root of Eq. (2.74). In the second equality we used the binomial theorem of Eq. (2.76), asssuming that $r > r_0$. In the third equality we plugged in our definition of α from Eq. (2.75). In the fourth equality we expanded out the parentheses and grouped terms according to the power of r_0/r. We then notice that the terms inside the square brackets are given by a power of r_0/r times a *polynomial* of $\cos\theta_0$. As it so happens, this identification of the coefficients is precisely the way Legendre introduced in 1782 what are now known as *Legendre polynomials*.[32] In short, we have arrived at the following remarkable result:

$$\frac{1}{|\mathbf{r}-\mathbf{r}_0|} = \frac{1}{r}\sum_{n=0}^{\infty}\left(\frac{r_0}{r}\right)^n P_n(\cos\theta_0) \tag{2.78}$$

[32] Legendre was studying the Newton potential, three years before Coulomb published his law.

where the P_n are the Legendre polynomials. Specifically, putting the last two equations together we find the first few Legendre polynomials:

$$\begin{aligned} P_0(x) &= 1, \qquad P_1(x) = x, \qquad P_2(x) = \frac{1}{2}(3x^2 - 1), \\ P_3(x) &= \frac{1}{2}(5x^3 - 3x), \qquad P_4(x) = \frac{1}{8}(35x^4 - 30x^2 + 3) \end{aligned} \tag{2.79}$$

For n odd the polynomial is an odd function of x and, similarly, for n even the polynomial is an even function of x. Higher-order polynomials can be derived analogously, by employing the binomial theorem, expanding the parentheses, and grouping terms.

Since this is a chapter on numerics, it's worth noticing that what we've accomplished with Eq. (2.78) is to trade a subtraction (which sometimes leads to catastrophic cancellation) on the left-hand side, for a sum of contributions on the right-hand side. Even if the signs oscillate, we are organizing our terms hierarchically.

A further point: in the derivation that led to our main result we assumed $r > r_0$. If we had been faced with a situation where $r < r_0$, we would have carried out an expansion in powers of r/r_0, instead, and the right-hand side would look similar to Eq. (2.78) but with the r and r_0 changing roles. In our specific example, we will be studying positions $\mathbf{r}$ away from the 36-charge array, so we will always be dealing with $r > r_0$.

Legendre Polynomials: from the Generating Function to Recurrence Relations

While you could, in principle, generalize the power-expansion approach above to higher orders (or even a general n-th order), it's fair to say that the manipulations become unwieldy after a while. In this subsection, we will take a different approach, one that also happens to be easier to implement programmatically. Let's take our main result of Eq. (2.78) and plug in the second equality of Eq. (2.74). We find:

$$\frac{1}{\sqrt{1 - 2\left(\frac{r_0}{r}\right)\cos\theta_0 + \left(\frac{r_0}{r}\right)^2}} = \sum_{n=0}^{\infty} \left(\frac{r_0}{r}\right)^n P_n(\cos\theta_0) \tag{2.80}$$

If we now define $u \equiv r_0/r$ and $x \equiv \cos\theta_0$, our equation becomes:

$$\frac{1}{\sqrt{1 - 2xu + u^2}} = \sum_{n=0}^{\infty} u^n P_n(x) \tag{2.81}$$

This is merely a reformulation of our earlier result. In short, it says that the function on the left-hand side, when expanded in powers of u, has coefficients that are the Legendre polynomials. As a result, the function on the left-hand side is known as the *generating function* of Legendre polynomials.

At this point we haven't actually gained anything from our re-definitions in terms of x

and u. To see the benefit, we differentiate Eq. (2.81) with respect to u and find:

$$\frac{x-u}{(1-2xu+u^2)^{3/2}} = \sum_{n=0}^{\infty} nP_n(x)u^{n-1} \tag{2.82}$$

If we now identify the $1/\sqrt{1-2xu+u^2}$ from Eq. (2.81) and move the remaining factor of $1/(1-2xu+u^2)$ to the numerator, we get:

$$(1-2xu+u^2)\sum_{n=0}^{\infty} nP_n(x)u^{n-1} + (u-x)\sum_{n=0}^{\infty} P_n(x)u^n = 0 \tag{2.83}$$

where we also moved everything to the same side. We now expand the parentheses and end up with five separate summations:

$$\sum_{n=0}^{\infty} nP_n(x)u^{n-1} - \sum_{n=0}^{\infty} 2nxP_n(x)u^n + \sum_{n=0}^{\infty} nP_n(x)u^{n+1} + \sum_{n=0}^{\infty} P_n(x)u^{n+1} - \sum_{n=0}^{\infty} xP_n(x)u^n = 0 \tag{2.84}$$

We are faced with a power series in u (actually a sum of five power series in u) being equal to zero, regardless of the value of u. This implies that the coefficient of each power of u (separately) is equal to zero. Thus, if we take a given power to be j (e.g., $j = 17$) our five summations above give for the coefficient of u^j the following:

$$(j+1)P_{j+1}(x) - 2jxP_j(x) + (j-1)P_{j-1}(x) + P_{j-1}(x) - xP_j(x) = 0 \tag{2.85}$$

which, after some trivial re-arrangements, gives us:

$$P_{j+1}(x) = \frac{(2j+1)xP_j(x) - jP_{j-1}(x)}{j+1} \tag{2.86}$$

To step through this process, we start with the known first two polynomials, $P_0(x) = 1$ and $P_1(x) = x$, and calculate $P_n(x)$ by taking:

$$j = 1, 2, \ldots, n-1 \tag{2.87}$$

This is known as *Bonnet's recurrence relation*. It's similar in spirit to Eq. (2.64), but here we won't face as many headaches. We will implement our new relation in Python in the following section. For now, we carry out, instead, the simpler task of picking up where Eq. (2.79) had left off: we plug $j = 4$ into Eq. (2.86) to find:

$$P_5(x) = \frac{9xP_4(x) - 4P_3(x)}{5} = \frac{1}{8}\left(63x^5 - 70x^3 + 15x\right) \tag{2.88}$$

In principle, this approach can be followed even for large n values.

At this point, we could turn to deriving a relation for the first derivative of Legendre polynomials. It's pretty easy to derive a recurrence relation that does this: as you will find out in a guided problem at the end of the chapter, instead of differentiating Eq. (2.81) with respect to u (as we did above), we could now differentiate that equation with respect to x. What this gives us is a recurrence relation for $P'_j(x)$. As the problem shows, we can do

even better, deriving a formula that doesn't necessitate a separate recurrence process for the derivatives. This is:

$$P'_n(x) = \frac{nP_{n-1}(x) - nxP_n(x)}{1 - x^2} \tag{2.89}$$

This equation only needs access to the last two Legendre polynomials, $P_n(x)$ and $P_{n-1}(x)$, which we produced when we were stepping through Bonnet's formula. We are now ready to implement both Eq. (2.86) and Eq. (2.89) in Python.

Implementation

Code 2.8 is an implementation of our two main relations for the Legendre polynomials and their derivatives. The function `legendre()` basically implements Eq. (2.86). We observe, however, that the first polynomial that that equation produces is $P_2(x)$: to evaluate $P_{j+1}(x)$ you need $P_j(x)$ and $P_{j-1}(x)$. Thus, we have hard-coded two special cases that simply spit out $P_0(x)$ and $P_1(x)$ (as well as $P'_0(x)$ and $P'_1(x)$) if the input parameter `n` is very small. All other values of `n` are captured by the `else`, which contains a loop stepping through the values of j given in Eq. (2.87). Note a common idiom we've employed here: we don't refer to $P_{j-1}(x)$, $P_j(x)$, and $P_{j+1}(x)$ but to `val0`, `val1`, and `val2`. There's no need to store the earlier values of the polynomial, so we don't.[33] Note that each time through the loop we have to change the interpretation of `val0`, `val1`, and `val2`: we do this via an idiomatic swap of two variables. Once we're done iterating, we turn to the derivative. As discussed around Eq. (2.89), this is not a recurrence relation: it simply takes in the last two polynomials and outputs the value of the derivative of the last Legendre polynomial. Our function returns a tuple of values: for a given n and a given x, we output $P_n(x)$ and $P'_n(x)$.[34]

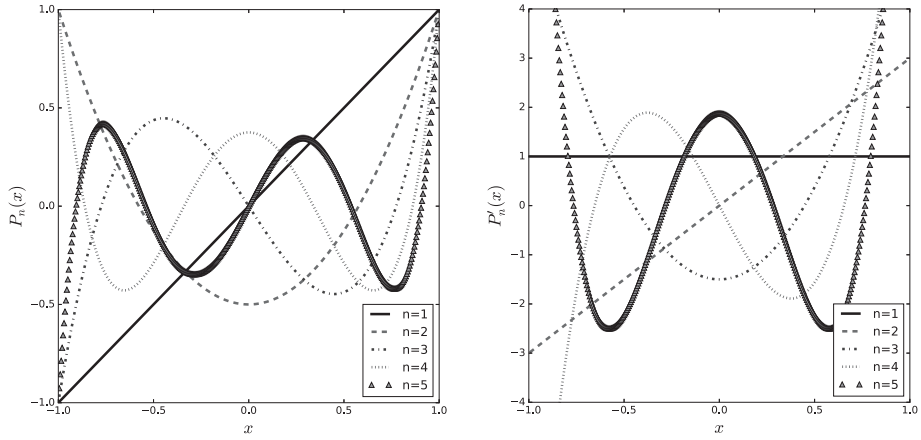
For most of this project, we provide the Python code that *could* be used to produce the figures we show, but we don't actually show the `matplotlib` calls explicitly. This is partly because it's straightforward to do so and partly because it would needlessly lengthen the size of this book. However, in the present case we've made an exception: as mentioned in the previous chapter, we follow the principle of "separation of concerns" and place all the plotting-related infrastructure in one function. The function `plotlegendre()` is used to plot five Legendre polynomials and their derivatives. These are 10 functions in total: as you can imagine, plotting so many different entities can become confusing if one is not careful. We have therefore taken the opportunity to show that Python dictionaries can also be helpful in tackling such mundane tasks as plotting. The first dictionary we employ determines the y-axis label: since we will be producing one plot for the Legendre polynomials and another plot for the Legendre-polynomial derivatives, we want these to have different

[33] Of course, we *do* calculate those, only to throw them away: another option would have been to also return all the $P_j(x)$'s to the user, since we get them "for free".

[34] You might need the derivative of Legendre polynomials if you wish to evaluate more involved electrostatic properties. We won't actually be doing that in what follows, but we *will* make use of the derivatives in chapter 7, when we discuss Gauss–Legendre integration.

Code 2.8 legendre.py

```
import matplotlib.pyplot as plt

def legendre(n,x):
     if n==0:
          val2 = 1.
          dval2 = 0.
     elif n==1:
          val2 = x
          dval2 = 1.
     else:
          val0 = 1.; val1 = x
          for j in range(1,n):
               val2 = ((2*j+1)*x*val1 - j*val0)/(j+1)
               val0, val1 = val1, val2
          dval2 = n*(val0-x*val1)/(1.-x**2)
     return val2, dval2

def plotlegendre(der,nsteps):
     plt.xlabel('$x$', fontsize=20)

     dertostr = {0: "$P_n(x)$", 1: "$P_n'(x)$"}
     plt.ylabel(dertostr[der], fontsize=20)

     ntomarker = {1: 'k-', 2: 'r--', 3: 'b-.', 4: 'g:', 5: 'c^'}
     xs = [i/nsteps for i in range (-nsteps+1,nsteps)]
     for n,marker in ntomarker.items():
          ys = [legendre(n,x)[der] for x in xs]
          labstr = 'n={0}'.format(n)
          plt.plot(xs, ys, marker, label=labstr, linewidth=3)

     plt.ylim(-3*der-1, 3*der+1)
     plt.legend(loc=4)
     plt.show()

if __name__ == '__main__':
     nsteps = 200
     plotlegendre(0,nsteps)
     plotlegendre(1,nsteps)
```



Legendre polynomials (left panel) and their first derivatives (right panel) **Fig. 2.6**

labels.[35] We then use a second dictionary, to encapsulate the correspondence from n value to line or marker style: since we're plotting five different functions (each time), we need to be able to distinguish them from each other. Once again, a moment's thought would show that the alternative is to explicitly build the `ys` and then call `plot()` for each function separately.[36] In contradistinction to this, we employ a loop over the dictionary items, and use fancy-string formatting to ensure that each curve label employs the correct number. Also, we use the parameter `der` (which helped us pick the right y-axis label) to select the appropriate element from the tuple returned by `legendre()`. Finally, we use `ylim()` to ensure that both plots look good. The result of running this code is shown in Fig. 2.6. Note that, due to their simplicity, we are not showing $P_0(x) = 1$ and $P'_0(x) = 0$.

2.5.3 Expansion for Many Point Charges

We are now ready to fuse the results of the previous two subsections: this means employing the multipole expansion for the case of our charge array. Before we do that, though, let's first write out the relevant equations explicitly.

Generalization

Earlier, we were faced with the potential coming from a single charge q_0, Eq. (2.73); we expanded the denominator and arrived at an equation in terms of Legendre polynomials, namely Eq. (2.78). In the present case, we would like to study the more general potential coming from n charges, as per Eq. (2.72):

$$\phi(\mathbf{r}) = \sum_{i=0}^{n-1} \phi_i(\mathbf{r}) = \sum_{i=0}^{n-1} k \frac{q_i}{|\mathbf{r} - \mathbf{r}_i|} \tag{2.90}$$

[35] Take a moment to think of other possible solutions to this task: the one that probably comes to mind first is to copy and paste. Most of the time, this is a bad idea.

[36] But what happens if you want to add a couple more functions to the plot?

If we carry out an expansion like that in Eq. (2.78) for each of these denominators, we get:

$$\begin{aligned}\phi(\mathbf{r}) &= \sum_{i=0}^{n-1}\frac{1}{r}\sum_{n=0}^{\infty}kq_i\left(\frac{r_i}{r}\right)^n P_n(\cos\theta_i) = k\sum_{n=0}^{\infty}\frac{1}{r^{n+1}}\sum_{i=0}^{n-1}q_i r_i^n P_n(\cos\theta_i)\\ &= \frac{k}{r}\sum_{i=0}^{n-1}q_i + \frac{k}{r^2}\sum_{i=0}^{n-1}q_i r_i\cos\theta_i + \frac{k}{r^3}\sum_{i=0}^{n-1}q_i r_i^2\frac{1}{2}\left(3\cos^2\theta_i - 1\right) + \cdots \end{aligned} \tag{2.91}$$

where θ_i is the angle between the vectors $\mathbf{r}$ and $\mathbf{r}_i$. In the second equality we interchanged the order of the summations and pulled out the denominators. In the third equality we wrote out the first few terms of the n summation and also plugged in the relevant $P_n(x)$ from Eq. (2.79).

At this point, we can spell out what was only implicit before: our approximation for the total potential at point $\mathbf{r}$ is given as a sum of terms of the form $1/r$, $1/r^2$, $1/r^3$, and so on:

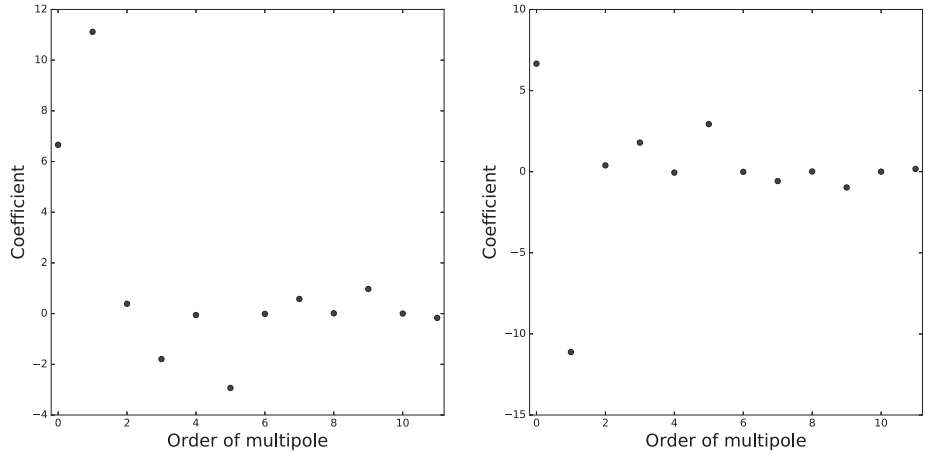
$$\phi(\mathbf{r}) = \frac{k}{r}Q_1 + \frac{k}{r^2}Q_2 + \frac{k}{r^3}Q_3 + \cdots \tag{2.92}$$

with the coefficients Q_j being a (possibly complicated) combination of the q_i, r_i, and θ_i. Even if you haven't encountered this specific problem in a course on electromagnetism, the general principle should be familiar to you: the terms we are now dealing with have names such as *monopole*, *dipole*, *quadrupole*, etc. In general, what we're carrying out is known as the *multipole expansion*. The monopole coefficient Q_1 is simply a sum over all the charges; the corresponding contribution to the potential goes as $1/r$, just like in the case of a single point charge. The dipole coefficient Q_2 is a sum over (q_i times) $r_i\cos\theta_i$, which can be re-expressed as $\hat{\mathbf{r}}\cdot\mathbf{r}_i$. Employing exactly the same argument, we see that the quadrupole coefficient Q_3 is a sum over q_i times $[3(\hat{\mathbf{r}}\cdot\mathbf{r}_i)^2 - r_i^2]/2$. Note that these coefficients may depend on the *direction* of $\mathbf{r}$, but not on its magnitude: all the dependence on the magnitude of $\mathbf{r}$ has been pulled out and is in the denominator.[37]

It's worth pausing for a moment to appreciate what we've accomplished in Eq. (2.92): by expanding in r_i/r (assuming, for now, that $r > r_i$) and interchanging the sums over n and i, we've expressed the full potential $\phi(\mathbf{r})$ (which we know from Eq. (2.72) is generally a complicated function of $\mathbf{r}$) as an expansion in powers of $1/r$, where the coefficients depend on the point charges, as well as the angles between $\mathbf{r}$ and $\mathbf{r}_i$. It goes without saying that increasingly large powers of $1/r$ have less and less of a role to play; of course, this also depends on the precise value of the coefficients. Thus, there are situations where the octupole, hexadecapole, and higher-order terms may need to be explicitly taken into account. Even so, what we've managed to do is to take a (possibly very complicated) distribution of point charges and encapsulate its effects on the total potential (along a given direction) into a few numbers, the Q_j.

Let's try to see the above insights applied to a specific case, that of our 36-charge array:

[37] Incidentally, moving the origin of the coordinates may change the precise values of Q_j, but the overall interpretation remains the same.



Multipole coefficients along the positive (left) and negative (right) y axis

Fig. 2.7

Fig. 2.7 shows the coefficients along the y axis.[38] As already mentioned, the coefficients Q_j don't depend on the precise point on the y axis, only on whether $\mathbf{r}$ is pointing up or down. Focusing on the positive y axis for the moment (left panel), we find that for this specific scenario the coefficients exhibit interesting structure: the monopole coefficient has a large magnitude, but the dipole coefficient is even larger.[39] After that, the even n coefficients seem to be pretty small, but the $n = 3, 5, 7, 9$ coefficients are sizable (and of both signs).

Still on the subject of our 36-charge array, let's now examine how the different coefficients are combined together. As you can see from Eq. (2.92), Q_j's of oscillating sign and large magnitude end up giving positive and negative contributions: however, as noted in our discussion near Eq. (2.78), these contributions are not of comparable magnitude, since they are multiplying powers of $1/r$. Thus, assuming r is reasonably large, the higher-order terms don't contribute too much. To reverse this argument, as you make r comparable to r_i you will need to keep track of an increasing number of terms in your multipole expansion. This is precisely what we find in the left panel of Fig. 2.8 (using a symmetrical-log scale): when you're close to the point charges the contributions mimic the size of the coefficients, but as you make r larger you need considerably fewer terms in the multipole expansion.[40]

The right panel of Fig. 2.7 shows the coefficients along the negative y axis: we notice that even-n multipole coefficients are unchanged, whereas odd-n multipole coefficients have a sign flipped. Taking the most prominent example, the electric dipole term is the largest in magnitude for both cases, but is positive in the left panel and negative in the right panel (this is easy to understand: $\hat{\mathbf{r}} \cdot \mathbf{r}_i$ changes sign when you flip the direction of $\hat{\mathbf{r}}$). The contributions in the right panel of Fig. 2.8 behave as we would expect.

[38] We picked the y axis for our $\mathbf{r}$'s in order to be consistent with what we showed in Fig. 2.5.
[39] Coefficients at different orders have different units.
[40] All contributions have the same units, C/m.

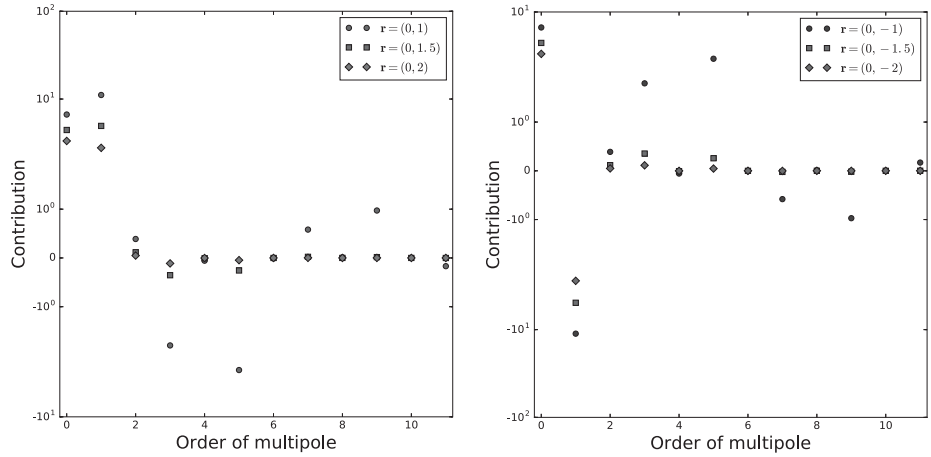


Fig. 2.8 Multipole contributions along the positive (left) and negative (right) y axis

Implementation

In order to produce Fig. 2.7 on the coefficients and Fig. 2.8 on the different contributions in the multipole expansion, we needed to make use of a code implementing the formalism we've introduced thus far. Code 2.9 is a Python implementation, in which we have separated out the different aspects of the functionality: (a) decomposition into magnitudes and angles, (b) the coefficients, and (c) combining the coefficients together with the value of r to produce an approximation to the full potential. This is the first major project in this book. While the program doesn't look too long, you notice that it starts by `importing` functionality from our earlier codes: `kahansum()` to carry out the compensated summation, `chargearray()` to produce our 36-charge array, `vecmag()` as a helper function, as well as `legendre()` to compute the Legendre polynomials that we need.

Our first new function, `decomp()`, takes in two vectors ($\mathbf{r}$ and $\mathbf{r}_i$, given as lists containing their Cartesian components), and evaluates their magnitudes as well as the cosine of the angle between them. As is common with our programs, there are several edge cases that this function does *not* cover: there will be a division failing if either of $\mathbf{r}$ and $\mathbf{r}_i$ is placed at the origin.[41] Another issue with this function is that it is wasteful: it evaluates `rmag` each time it is called (i.e., 36 times in our case), even though the input `rs` hasn't changed: as usual, this is because we've opted in favor of code simplicity.

Our next function, `multicoes()`, is a straightforward implementation of Eq. (2.92) or, if you prefer, of the second equality in Eq. (2.91). It takes in as parameters `qtopos` (containing the distribution of point charges), `rs` (this being the position $\mathbf{r}$ at which we

[41] While we're taking $r > r_i$, it's possible that our charge array could contain a charge at the coordinate origin.

multipole.py Code 2.9

```
from kahansum import kahansum
from chargearray import chargearray, vecmag
from legendre import legendre

def decomp(rs,ris):
     rmag = vecmag(rs); rimag = vecmag(ris)
     prs = [r*ri for r,ri in zip(rs,ris)]
     vecdot = kahansum(prs)
     costheta = vecdot/(rmag*rimag)
     return rmag, rimag, costheta

def multicoes(rs,qtopos,nmax=60):
     coes = [0. for n in range(nmax+1)]
     for n in range(nmax+1):
          for q,pos in qtopos.items():
               rmag, rimag, costheta = decomp(rs,pos)
               val = q*(rimag**n)*legendre(n,costheta)[0]
               coes[n] += val
     return coes

def multifullpot(rs,qtopos):
     coes = multicoes(rs,qtopos)
     rmag = vecmag(rs)
     contribs = [coe/rmag**(n+1) for n,coe in enumerate(coes)]
     return kahansum(contribs)

if __name__ == '__main__':
     qtopos = chargearray(6)
     for y in 1,-1:
          rs = [0.,y]
          potval = multifullpot(rs,qtopos)
          print(rs, potval)
```

are interested in evaluating/approximating the potential), and `nmax` (which is a parameter controlling how many terms we should keep in the multipole expansion). We provide a default parameter value for `nmax` which is presumably large enough that we won't have to worry about the quality of our approximation. A problem asks you to investigate this in detail, for obvious reasons: why use a large number of terms in the multipole expansion

if only a handful are enough for your purposes? The structure of our function closely follows that of Eq. (2.91): the loop over n is outside and that over i inside (though, as earlier, there's no need to carry around an explicit i index). We use the `items()` method of Python dictionaries (which you should have gotten used to by now) to step through all the q_i's in our distribution. The only subtlety is that our call to `legendre()` ends with `[0]`: this results from the fact that `legendre()` returns a tuple of two numbers. While `multicoes()` is a pretty function, note that it (and the rest of the code) *assumes* that $r > r_i$ is true: a problem asks you to remove this assumption, rewriting the code appropriately.

Our last function, `multifullpot()`, is essentially a test: it computes the multipole-expansion approximation of Eq. (2.92) for the total potential $\phi(\mathbf{r})$, which can be compared to the complete calculation of it in Eq. (2.72). Note that the philosophy here is to sum the contribution from each multipole (with the coefficient for each multipole term coming from all the q_i charges), whereas the philosophy of `fullpot()` in our earlier code was to simply sum together the full contributions from each q_i, as per Eq. (2.72). This function is, as usual, idiomatic (employing a list comprehension, as well as `enumerate()`). At the end, it sums together (using `kahansum()`) all the contributions: this is different from Fig. 2.8, which shows each contribution separately.

The main program again picks two arbitrary points on the y axis and prints out the multipole-expansion approximation to the value of the potential $\phi(\mathbf{r})$ at those points. Comparing those numbers to the output of our earlier code `chargearray.py`, we see that they are in pretty good agreement with the full potential values. This isn't too surprising, since we picked `nmax=60` as our default parameter value.[42] The results may feel somewhat underwhelming: this impression can be altered if you recall that this code can (be modified in order to) produce Fig. 2.7 on the coefficients and Fig. 2.8 on the different contributions in the multipole expansion. These figures can help you build insight about the specific geometry involved in our 36-charge array and on the role of the different multipole terms.

2.6 Problems

1. Using the notation we employed in the main text, the absolute value of the relative error is $|\delta x| = |(\tilde{x} - x)/x|$. In practice, it is sometimes convenient to provide a bound for a distinct quantity, $|\tilde{\delta} x| = |(\tilde{x} - x)/\tilde{x}|$. Using these two definitions, try to find an inequality relating $|\tilde{\delta} x|$ and $|\delta x|$ (i.e., $A \leq |\tilde{\delta} x| \leq B$, where A and B contain $|\delta x|$).
2. Study the propagation of the relative error, for the case of division, $x = a/b$, by analogy to what was done in the main text for the case of multiplication.
3. This problem studies error propagation for cases that are more involved than what we encountered in the main text. Specifically, find the:

   (a) Absolute error in $y = \ln x$.
   (b) Relative error in $y = \sqrt{x}$.

[42] Given that our points are pretty close to the charge array, this was probably a safe choice.

4. This problem studies overflow in Python in more detail.

   (a) Modify the code in section 2.3.2 to investigate underflow for floating-point numbers in Python. In order to make your output manageable, make sure you start from a number that is sufficiently small.

   (b) Now investigate whether or not overflow occurs for integers in Python. Do yourself a favor and start not from 1, but from a very large positive number and increment from there. You should use `sys.maxsize` to get the ballpark of near where you should start checking things. You should use `type()` to see if the type changes below `sys.maxsize` and above it.

5. Run the following code and see what happens:

```
from math import sqrt

def f(x,nmax=100):
      for i in range(nmax):
            x = sqrt(x)
      for i in range(nmax):
            x = x**2
      return x

for xin in (5., 0.5):
      xout = f(xin)
      print(xin, xout)
```

   Without rounding error, we would expect the output to be $f(x) = x$ (we take the square root a number of times and then take the square the same number of times, so nothing happens). Note that this code does *not* involve any subtractions whatsoever, so the "fear" of catastrophic cancellation plays no role here. Dig deeper, to determine why you get the output you get. Once you've uncovered the issue, you'll understand why, at the start of section 2.4 we mentioned one or two iterations being the culprits.

6. We now examine a case where plotting a function on the computer can seriously mislead us. The function we wish to plot is: $f(x) = x^6 + 0.1 \log(|1 + 3(1 - x)|)$. Use 100 points from $x = 0.5$ to 1.5 to plot this function in `matplotlib`. Do you see a dip? Consider the function itself and reason about what you should be seeing. Then use a much finer grid and ensure that you capture the analytically expected behavior.

7. Take the standard quadratic equation:

$$ax^2 + bx + c = 0 \tag{2.93}$$

   The formula for the solutions of this equation is very well known:

$$x_{\pm} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{2.94}$$

Take $b > 0$ for concreteness. It is easy to see that when $b^2 \gg ac$ we don't get a catastrophic cancellation when evaluating $b^2 - 4ac$ (we may still get a "benign" cancellation). Furthermore, $\sqrt{b^2 - 4ac} \approx b$. However, this means that x_+ will involve catastrophic cancellation in the numerator.

We will employ an analytical trick in order to help us preserve significant figures. Observe that the product of the two roots obeys the relation:

$$x_+ x_- = \frac{c}{a} \tag{2.95}$$

The answer now presents itself: use Eq. (2.94) to calculate x_-, for which no catastrophic cancellation takes place. Then, use Eq. (2.95) to calculate x_+. Notice that you ended up calculating x_+ via division only (i.e., without a catastrophic cancellation). Write a Python code that evaluates and prints out: (a) x_-, (b) x_+ using the "bad" formula, and (c) x_+ using the "good" formula. Take $a = 1, c = 1, b = 10^8$. Discuss the answers.[43]

8. We promised to return to the distinction between catastrophic and benign cancellation. Take $\tilde{x}$ and $\tilde{y}$ to be: $\tilde{x} = 1234567891234567.0$ and $\tilde{y} = 1234567891234566.0$. Now, if we try to evaluate $\tilde{x}^2 - \tilde{y}^2$ we will experience catastrophic cancellation: each of the squaring operations leads to a rounding error and then the subtraction exacerbates that dramatically. Write a Python code that does the following:

   (a) Carries out the calculation $1234567891234567^2 - 1234567891234566^2$ using integers, i.e., exactly.
   (b) Carries out the subtraction $1234567891234567.0^2 - 1234567891234566.0^2$ using floats, i.e., exhibiting catastrophic cancellation.
   (c) Now, we will employ a trick: $x^2 - y^2$ can be re-expressed as $(x-y)(x+y)$. Try using this trick for the floats and see what happens. Does your answer match the integer answer or the catastrophic-cancellation answer? Why?

9. We will study the following function:

$$f(x) = \frac{1 - \cos x}{x^2} \tag{2.96}$$

   (a) Start by plotting the function, using a grid of the form $x = 0.1 \times i$ for $i = 1, 2, \ldots, 100$. This should give you some idea of the values you should expect for $f(x)$ at small x.
   (b) Verify your previous hunch by taking the limit $x \to 0$ and using L'Hôpital's rule.
   (c) Now, see what value you find for $f(x)$ when $x = 1.2 \times 10^{-8}$. Does this make sense, even qualitatively?
   (d) Use a trigonometric identity that enables you to avoid the cancellation. Evaluate the new function at $x = 1.2 \times 10^{-8}$ and compare with your analytical answer for $x \to 0$.

10. This problem focuses on analytical manipulations introduced in order to avoid a cancellation. Rewrite the following expressions in order to evaluate them for large x:

    (a) $\sqrt{x+1} - \sqrt{x}$

[43] Keep in mind that if $b^2 \approx ac$, then $b^2 - 4ac$ would involve a catastrophic cancellation. Unfortunately, there is no analytical trick to help us get out of this problem.

(b) $\dfrac{1}{x+1} - \dfrac{2}{x} + \dfrac{1}{x-1}$

(c) $\dfrac{1}{\sqrt{x}} - \dfrac{1}{\sqrt{x+1}}$

Feel free to test the "before" and "after" formulas with Python.

11. We'll spend more time on statistical concepts in chapters 6 and 7 but, for now, let's focus on numerical-error aspects. Assume you have n values x_i. First, evaluate the mean:

$$\mu = \frac{1}{n}\sum_{i=0}^{n-1} x_i \tag{2.97}$$

You can evaluate the variance using a *two-pass* algorithm:

$$\sigma^2 = \frac{1}{n}\sum_{i=0}^{n-1}(x_i - \mu)^2 \tag{2.98}$$

This is called a two-pass algorithm because you need to evaluate the mean first, so you have to loop through the x_i once to get the mean and a second time time to get the variance. Many people prefer the following *one-pass* algorithm:

$$\sigma^2 = \left(\frac{1}{n}\sum_{i=0}^{n-1} x_i^2\right) - \mu^2 \tag{2.99}$$

You should be able to see that this formula allows you to keep running sums of the x_i and the x_i^2 values in parallel and then perform only one subtraction at the end.

Naively, you might think that the two-pass algorithm will suffer from more roundoff error problems, since it involves n subtractions. On the other hand, if you solved the earlier problem on $\tilde{x}^2 - \tilde{y}^2$, you might be more wary of subtracting the squares of two nearly equal numbers (which is what the one-pass algorithm does). Write two Python functions, one for each algorithm, and test them on the two cases below:

$$\begin{aligned} x_i &= 0, 0.01, 0.02, \ldots, 0.09 \\ x_i &= 123456789.0, 123456789.01, 123456789.02, \ldots, 123456789.09 \end{aligned} \tag{2.100}$$

12. This problem discusses error buildup when trying to evaluate polynomials without and with the use of *Horner's rule*. Take a polynomial of degree $n-1$:

$$P(x) = p_0 + p_1 x + p_2 x^2 + p_3 x^3 + \cdots + p_{n-1} x^{n-1} \tag{2.101}$$

Write a function that takes in a list containing the p_i's, say `coeffs`, and the point x and evaluates the value of $P(x)$ in the naive way, i.e., from left to right.

Notice that this way of coding up the polynomial contains several (needless) multiplications. This is so because x^i is evaluated as $x \times x \times \cdots \times x$ (where there are $i-1$ multiplications). Thus, this way of approaching the problem corresponds to:

$$1 + 2 + \cdots + n - 2 = \frac{(n-1)(n-2)}{2} \tag{2.102}$$

multiplications, from x^2 all the way up to x^{n-1}. If we rewrite the polynomial:

$$P(x) = p_0 + x\,(p_1 + x\,(p_2 + x\,(p_3 + \cdots + x\,(p_{n-2} + x p_{n-1})\cdots))) \tag{2.103}$$

then we can get away with only $n - 1$ multiplications (i.e., no powers are evaluated). This is obviously more efficient, but equally important is the fact that this way we can substantially limit the accumulation of rounding error (especially for polynomials of large degree). Write a function that takes in a list containing the p_i's, say `coeffs`, and the point x and evaluates the value of $P(x)$ in the new way, i.e., from right to left.

Apply the previous two functions to the (admittedly artificial) case of:

```
coeffs = [(-11)**i for i in reversed(range(8))]
```

and $x = 11.01$. Observe any discrepancy and discuss its origin.

13. This problem studies the rational function introduced in the main text, Eq. (2.53).

    (a) Apply Horner's rule twice (once for the numerator and once for the denominator) to produce two plots, one for $x = 1.606 + 2^{-52}i$ and one for $x = 2.400 + 2^{-52}i$. Your results should look like Fig. 2.3.

    (b) Create a new Python function that codes up the following expression:

    $$s(x) = 4 - \frac{3(x-2)[(x-5)^2 + 4]}{x + (x-2)^2[(x-5)^2 + 3]} \tag{2.104}$$

    which is a rewritten version of our rational function. Apply this new function to the case of the previous two plots and compare the rounding error pattern, size, etc.

    (c) Now introduce two more sets of results, this time for the starting expression for $r(x)$ produced using (not Horner's rule but) the naive implementation, using powers (i.e., the way you would have coded this up before doing the previous problem). Interpret your findings.

14. This problem studies a new rational function:

    $$t(x) = \frac{7x^4 - 101x^3 + 540x^2 - 1204x + 958}{x^4 - 14x^3 + 72x^2 - 151x + 112} \tag{2.105}$$

    Notice that the denominator is the same as in the previous problem.

    (a) Plot $t(x)$, evaluated via Horner's rule, along with the following (equivalent) continued fraction, from $x = 0$ to $x = 4$:

    $$u(x) = 7 - \cfrac{3}{x - 2 - \cfrac{1}{x-7+\cfrac{10}{x-2-\cfrac{2}{x-3}}}} \tag{2.106}$$

    You may wish to know that:

    $$u(1) = 10,\ u(2) = 7,\ u(3) = 4.6,\ u(4) = 5.5 \tag{2.107}$$

    (b) Evaluate each of these functions for $x = 10^{77}$ (make sure to use floats in your code). Do you understand what is happening? Are you starting to prefer one formulation over the other? (What happens if you use integers instead of floats?)

    (c) Plot the two Python functions ($t(x)$ and $u(x)$) for $x = 2.400 + 2^{-52}i$, where i goes from 0 to 800. Was your intuition (about which formulation is better) correct?

15. In the main text we investigated the Taylor series for e^x at $x = 0.1$ and $x = 20$. Do the same for $f(x) = \sin x$ at $x = 0.1$ and $x = 20$. Just like we did, you will first need to find a simple relation between the n-th term and the $(n-1)$-th term in the series. Then, use a trigonometric identity to study a smaller x instead of 20.
16. Here we study the *Basel problem*, namely the sum of the reciprocals of the squares of the positive integers. This also happens to be a specific value of the Riemann zeta function, $\zeta(2)$. The value can be calculated in a variety of ways and turns out to be:

$$\sum_{k=1}^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6} = 1.644\,934\,066\,848\,226\,4\ldots \tag{2.108}$$

Here we will use Python to approximate the sum numerically.

(a) Code this up in the obvious way, namely by adding all the contributions from $k = 1, 2$, up to some large integer. You can break out of the loop when the value of the sum stops changing. What is the value of the sum when this happens?

(b) Make sure to understand why the sum stopped changing when it did. Is there some meaning behind the value of the maximum integer (inverse squared) when this happens? Call this `nmaxd` for future reference.

(c) If you're disappointed in the number of matching significant digits we get this way, do not despair: we are summing the contributions from largest to smallest, so by the time we get to the really tiny contributions they stop mattering. The obvious fix here is to *reverse* the order in which we are carrying out the sum. This has the advantage of dealing with the tiniest contributions first (you may have encountered the term "subnormal numbers" in Appendix B). The disadvantage is that our previous strategy regarding termination (break out when the sum stops changing) doesn't apply here: when summing in reverse the last few contributions are the largest in magnitude.

What we do, instead, is simply pick a large maximum integer `nmaxr` from the start. Be warned that this part of the problem will start to take a long time, depending on the CPU you are using. Do a few runs for `nmaxr` a multiple of `nmaxd` (4, 8, 16, 32). You should see the answer get (slowly) better. The beauty here is that the larger `nmaxr`, the better we can do. (Try increasing `nmaxd` for the direct method and observe that it makes no difference.)

(d) Employ compensated summation to carry out the sum. Test the Kahan sum function out to see how well it does, despite the fact that it carries out the sum in direct order, i.e., by starting from the largest contribution. Notice that we don't have a nifty termination/breaking out criterion. On the other hand, by increasing the number of terms, we can still make a difference.

17. In section 2.4.5 we saw a case where backward recursion avoids the problems of forward recursion. Another example is provided by *spherical Bessel functions* (of the first kind):

$$j_{n+1}(x) + j_{n-1}(x) = \frac{2n+1}{x} j_n(x) \tag{2.109}$$

which is analogous to Bonnet's recurrence relation, Eq. (2.86). Evaluate $j_4(0.5)$ in the "naive way", starting from the known functions:

$$j_0(x) = \frac{\sin x}{x}, \qquad j_1(x) = \frac{\sin x}{x^2} - \frac{\cos x}{x} \tag{2.110}$$

and see what goes wrong. Then, use backward recursion starting from $n = 15$. Since your two starting guesses are arbitrary, you should normalize by computing:

$$j_4(0.5) = \tilde{j}_4(0.5)\frac{j_0(0.5)}{\tilde{j}_0(0.5)} \tag{2.111}$$

at the end, where $\tilde{j}_0(0.5)$ and $\tilde{j}_4(0.5)$ are the values you computed and $j_0(0.5)$ is the correctly normalized value from Eq. (2.110).

18. In this problem we will study the Fourier series of a periodic square wave. As you may already know, this gives rise to what is known as the *Gibbs phenomenon*, which is a "ringing" effect that appears at discontinuities. Specifically, from $-\pi$ to π we have:

$$f(x) = \begin{cases} \frac{1}{2}, & 0 < x < \pi \\ -\frac{1}{2}, & -\pi < x < 0 \end{cases} \tag{2.112}$$

The Fourier series of this function is:

$$f(x) = \frac{2}{\pi} \sum_{n=1,3,5,\ldots} \frac{\sin(nx)}{n} \tag{2.113}$$

(a) Create two Python functions, one for the square wave and another for its Fourier expansion. The latter should take in as an argument the maximum n up to which you wish the sum to go.

(b) Plot the square wave and the Fourier-expansion results for $n_{max} = 1, 3, 5, 7, 9$ (six curves in total). Where are the oscillation amplitudes largest?

(c) Note that this issue arises not from roundoff error, but from the nature of the Fourier series itself. To convince yourself that this is, indeed, the case, take the maximum n value to be 21, 51, 101, and so on. What do you find?

19. Using `vectorfield.py` and `chargearray.py`, produce panels analogous to those in Fig. 2.4, made up of 36 charges (along six rows and six columns). In this problem, in addition to changing the magnitude of the charges (to either $+1$ or -1) you should also change their placement (i.e., they no longer need to be alternating as in our figure). Separately investigate the cases where the field at large distances behaves like: (a) a dipole, and (b) a quadrupole.

20. An alternative recurrence relation for Legendre polynomials is:

$$P_n(x) = 2xP_{n-1}(x) - P_{n-2}(x) - \frac{xP_{n-1}(x) - P_{n-2}(x)}{n} \tag{2.114}$$

Compare with the output of `legendre.py`.

21. This problem guides you toward deriving our result for the first derivative of Legendre polynomials, Eq. (2.89).

(a) Differentiate Eq. (2.81) with respect to x. Then, split the resulting equation into four separate sums, noticing that our power series in u is equal to zero, regardless of the value of u. Thus, the coefficient of each power of u (separately) is equal to zero. This will be a recurrence relation for $P'_j(x)$ (since we already know the Legendre polynomials $P_j(x)$).
(b) Differentiate Eq. (2.86) with respect to x.
(c) Multiply the result of part (b) by 2 and add to that part (a) times $2j + 1$.
(d) Produce two equations, one being the sum of the results of part (a) and part (c) (divided by 2) and the other being the difference of the results of part (a) and part (c) (divided by 2). Call these two equations S and D, respectively.
(e) Take $j \to j + 1$ in S and subtract x times D. The result should be equivalent to Eq. (2.89).

22. This problem studies the evaluation of Bernoulli numbers and polynomials.

(a) In section 2.5.2 we saw how to go from a generating function to a recurrence relation for Legendre polynomials. We will now do something analogous for what are known as *Bernoulli numbers*. Specifically, start from:

$$\frac{u}{e^u - 1} = \sum_{j=0}^{\infty} \frac{B_j u^j}{j!} \tag{2.115}$$

and show that:

$$n - 1 = \sum_{j=1}^{n-1} B_{2j} \binom{2n}{2j} \tag{2.116}$$

You should use the Taylor series of the exponential and the Cauchy product.
(b) Compute the first 15 even Bernoulli numbers from the recurrence relation.
(c) Slightly generalizing the above generating function, we get to a corresponding relation for *Bernoulli polynomials*:

$$\frac{ue^{ut}}{e^u - 1} = \sum_{j=0}^{\infty} B_j(t) \frac{u^j}{j!} \tag{2.117}$$

Use this equation to derive the following properties:

$$B_j(0) = B_j, \qquad \frac{d}{dt} B_j(t) = jB_{j-1}(t), \qquad B_j(1) = (-1)^j B_j(0) \tag{2.118}$$

(d) We will now use the previous three properties to derive the celebrated *Euler–Maclaurin summation formula*. Since $B_0(t) = B_0 = 1$, we can write:

$$\int_0^1 g(t)dt = \int_0^1 g(t)B_0(t)dt \tag{2.119}$$

You can now use the second property and replace $B_0(t)$ with $B'_1(t)$. Integrate by parts and keep repeating the entire exercise until you find:

$$\int_0^1 g(t)dt = \frac{g(0)}{2} + \frac{g(1)}{2} - \sum_{j=1}^{m} \frac{1}{(2j)!} B_{2j} \left[g^{(2j-1)}(1) - g^{(2j-1)}(0) \right]$$

$$+\frac{1}{(2m)!}\int_0^1 g^{(2m)}(t)B_{2m}(t)dt \tag{2.120}$$

Crucially, only even Bernoulli numbers appear here.

(e) To make the integral and derivatives easier, take $g(t) = e^t$. Print out the value of the left-hand side in Eq. (2.120) as well as the value of the right-hand side as you take $m = 1, 2, 3, \ldots, 10$ (always dropping the remainder term).

23. This problem studies the evaluation of *Chebyshev polynomials* and their extrema, known as *Chebyshev points*. First, implement the following recurrence relation:

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x) \tag{2.121}$$

starting from the known functions $T_0(x) = 1$ and $T_1(x) = x$. Plot the first few Chebyshev polynomials from $x = -1$ to $x = +1$. Second, use trigonometric identities to show (analytically) that the representation:

$$T_n(x) = \cos\left(n\cos^{-1}x\right) \tag{2.122}$$

is equivalent to that in Eq. (2.121). Finally, use Eq. (2.122) to show that the n extrema of $T_{n-1}(x)$ are:

$$x_j = -\cos\left(\frac{j\pi}{n-1}\right), \qquad j = 0, 1, \ldots, n-1 \tag{2.123}$$

This result will play an important role in chapter 6.

24. For the 36-charge array of section 2.5 determine, as a function of position along the y axis, the minimum number of terms you need to keep in the multipole expansion in order to ensure your relative error in the total potential is less than 10^{-6}.

25. Generalize our code in `multipole.py` so that it works regardless of whether or not $r > r_i$. This necessitates changes to all three functions in that program.