Integrals 7

For human beings there is no greater evil than necessary chance.

Sophocles

7.1 Motivation

We now turn to *numerical integration*, also known as *quadrature*. At a big-picture level, it's good to keep in mind that in numerical integration "throwing more points at the problem" typically pays off; it is common that one can provide the answer for a definite integral (approximately) to within machine precision.

7.1.1 Examples from Physics

In the spirit of starting each chapter with appropriately motivated problems, we now discuss a few integrals that pop up in different areas of physics.

1. **Electrostatic potential away from a uniformly charged rod**
   Our first (quasi-random) example comes from classical electromagnetism. When calculating the electrostatic potential at a distance d away from (the start of) a finite uniformly charged rod, we are faced with the calculation:

$$V = \int \frac{kdq}{r} = \int_0^a \frac{k\lambda dx}{\sqrt{x^2 + d^2}} = \int_0^a \frac{k\lambda dx}{d\sqrt{\left(\frac{x}{d}\right)^2 + 1}} \tag{7.1}$$

   where λ is the linear density of electric charge and a is the length of the rod. Changing variables to $y = x/d$ and assuming $a/d = 1$, leads to the integral:

$$\Phi = \int_0^1 dx \frac{1}{\sqrt{x^2 + 1}} \tag{7.2}$$

   where we re-named the dummy integration variable y back to x. This integral can be evaluated analytically: substitute $x = \tan\theta$ and then take $w = \tan\theta + \sec\theta$. The answer turns out to be:

$$\Phi = \ln(x + \sqrt{x^2 + 1})\Big|_0^1 = \ln(1 + \sqrt{2}) \approx 0.881\,373\,587\,019\,542\ldots \tag{7.3}$$

   Here, the indefinite integral happens to be an inverse hyperbolic sine. We will employ this example in what follows, for the sake of concreteness.

2. **Maxwell–Boltzmann distribution for the velocities**
   While it's not totally trivial to come up with such substitutions, you could always use a symbolic math package or even a table of integrals. However, in some cases there may be no simple analytical answer. A well-known example appears in the theory of statistical mechanics. Take the Maxwell–Boltzmann distribution for the velocities of an ideal gas (in one dimension):

$$P(v_x) = \sqrt{\frac{\beta m}{2\pi}} e^{-\beta m v_x^2/2} \tag{7.4}$$

   where v_x is the velocity and the temperature is hidden inside $\beta = 1/k_B T$. If we're interested in the probability of particles having velocities from, say, $-A$ to A, then the integral we need to evaluate can be recast as:

$$I = \int_{-A}^{A} P(v_x) dv_x = \frac{1}{\sqrt{2\pi}} \int_{-A\sqrt{\beta m}}^{A\sqrt{\beta m}} e^{-x^2/2} dx \tag{7.5}$$

   The answer here cannot be expressed in terms of elementary functions (instead, this is precisely the definition of the error function, erf($A\sqrt{\beta m}$). This is important enough that it bears repeating: the indefinite integral of a simple Gaussian cannot be expressed in terms of elementary functions.[1]
3. **Statistical mechanics starting from elementary degrees of freedom**
   There is a case where simple analytical evaluation or standard numerical quadrature methods both fail: multidimensional integrals. For example, the (interaction part of the) classical partition function of a gas of n atoms is:

$$Z = \int d^3 r_0 d^3 r_1 \ldots d^3 r_{n-1} e^{-\beta V(\mathbf{r}_0, \mathbf{r}_1, \ldots, \mathbf{r}_{n-1})} \tag{7.6}$$

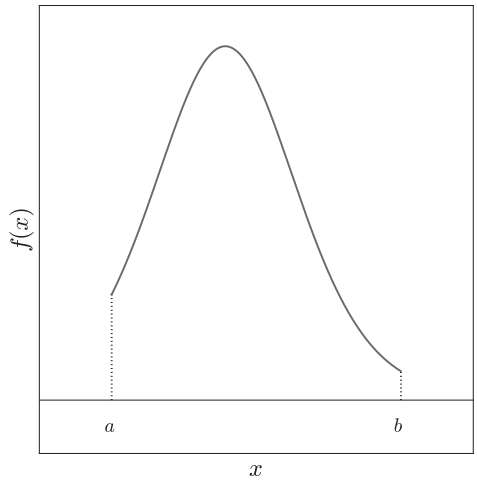
   Here, $\beta = 1/k_B T$ is the inverse temperature and $V(\mathbf{r}_0, \mathbf{r}_1, \ldots, \mathbf{r}_{n-1})$ contains the interactions between particles (or with external fields). This is clearly a $3n$-dimensional integral, evaluating which analytically (or via numerical quadrature) is hopeless for all but the smallest values of n.

7.1.2 The Problem to Be Solved

More generally, the task of quadrature is to evaluate $\int_a^b f(x)dx$, for a function $f(x)$ which may look like that in Fig. 7.1. Numerical integration at its most fundamental consists of approximating a definite integral by a sum, as follows:

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n-1} c_i f(x_i) \tag{7.7}$$

[1] If we take the limit $A \to \infty$, then we *can* carry out the integral analytically: this gives $\sqrt{2\pi}$ and therefore $I = 1$. This is a very important integral, which you will re-encounter in the problem set.



The definite integral is the area under the curve, when x goes from a to b **Fig. 7.1**

Here, the x_i are known as the *nodal abscissas* and the c_i are known as the *weights*. Note that the i runs from 0 to $n - 1$, so we are dealing with n abscissas (and n weights) in total.

Such quadrature methods (and much of this chapter) can be divided into two large categories. First, we have *closed methods*, where the endpoints of our interval (a and b) are included as abscissas (x_i). Of the approaches we discuss below, the trapezoid and Simpson's methods are closed. Second, we have *open methods*, where the endpoints of our interval (a and b) are *not* included as abscissas (x_i). Of the approaches we discuss below, the midpoint method and Gaussian quadrature are open. Most categorizations are too neat and this one is no exception: there exists a third category, namely the case of *half-open* methods, where one of the endpoints of our interval (a or b) is not included in the abscissas (x_i). Of the approaches we discuss below, the rectangle method is half-open. It's easy to see that open or half-open methods are to be preferred in the case where the integrand has a singularity at an endpoint.

Above, we mentioned several integration methods by name, grouping them according to whether or not they are closed, open, or half-open. In practice, it is much more common to group them, instead, according to a different criterion, namely whether they are:

- **Newton–Cotes methods**: these make the assumption that the integral can be approximated by summing up the areas of elementary shapes (e.g., rectangles); such methods typically involve *equally spaced abscissas*: these are useful (if not necessary) when $f(x)$ has already been evaluated at specific points on a grid.
- **Gaussian quadrature**: these make use of *unequally spaced abscissas*: these methods choose the x_i in such a way as to provide better accuracy. As a result, they typically require fewer abscissas and therefore fewer function evaluations, making them an attractive option when an $f(x)$ evaluation is costly.

Keep in mind that some of the Newton–Cotes methods we discuss below (e.g., the rectangle rule) don't, strictly speaking, require equally spaced abscissas (e.g., we could have adjacent

rectangles of different widths), but in all that follows we will only use points on an equally spaced grid for Newton–Cotes methods, reserving unequally spaced abscissas for Gaussian quadrature methods.

A large part of our discussion of Newton–Cotes and Gaussian quadrature methods will involve a study of the errors involved; this will allow us to introduce *adaptive integration* (section 7.3), wherein one quantifies the error in the integration even when the analytical answer is not known (which in practice is usually the case). A similar idea is involved in *Romberg integration* (section 7.4), which combines a low-order approach with Richardson extrapolation, to produce a more accurate answer. The second half of the chapter will focus on *multidimensional integration*, for which Newton–Cotes and even Gaussian quadrature methods are typically too inefficient. This will allow us to introduce the important subject of *Monte Carlo integration*, also known as *stochastic integration*, which employs random numbers; this is also used in the Project that rounds out this chapter.

You may be wondering why the present chapter, on integration, comes much later in the book than chapter 3 on numerical differentiation. The answer is that we will be using a lot of the machinery developed in earlier chapters, most notably our infrastructure on interpolation from chapter 6 and on root-finding from chapter 5. Since this is the penultimate chapter of the book, we assume the reader has developed some maturity from working through the earlier material. Similarly, our codes will use `numpy` functionality repeatedly, which we had opted against in chapter 3.

7.2 Newton–Cotes Methods

As already noted, Newton–Cotes methods evaluate the integral as a sum of elementary areas (rectangles, trapezoids, etc.). In this book, we use Newton–Cotes methods that employ an equally spaced grid: this is similar (nay, identical) to what we saw in chapter 3, when we examined derivatives for points on a grid in section 3.3.6. As a reminder, the assumption is that we have access to a set of n discrete data points (i.e., a table) of the form $(x_i, f(x_i))$ for $i = 0, 1, \ldots, n-1$. The points x_i are on an equally spaced grid, from a to b. The n points then are given by the following relation:

$$x_i = a + ih \tag{7.8}$$

where, as usual, $i = 0, 1, \ldots, n-1$. The h is clearly given by:

$$h = \frac{b-a}{n-1} \tag{7.9}$$

Recall that we are dealing with n points in total, so we are faced with $n-1$ subintervals from a to b. As always, we are using the terms *subinterval* and *panel* interchangeably. (A

final reminder, which will come in handy later in this chapter: you may want to introduce a new variable containing the number of panels, $N = n - 1$, in which case $h = (b - a)/N$; then, the i in x_i would go from 0 to N, since we have $N + 1 = n$ points in total.)

For each of the several methods covered here, we will first start out with a version of the answer for a small problem (e.g., for one panel, an approximation to the integral $\int_{x_i}^{x_{i+1}} f(x)dx$). This includes a graphic interpretation as well as an appropriately motivated formula. In each case, we then turn to what is known as the *composite* formula, which is nothing other than a sum of all the "small-problem" answers like $\int_{x_i}^{x_{i+1}} f(x)dx$ and is thereby an approximation to the integral $\int_a^b f(x)dx$. That is then followed by an expression for the approximation error made by employing that specific composite formula. In three cases (rectangle, trapezoid, Simpson's methods) we also provide a derivation of the expression for the error as well as a full Python implementation.

7.2.1 Rectangle Rule

We start with the one-panel version of the rectangle rule; it may help to periodically look at Fig. 7.2 while reading. For the one-panel version, we are interested in approximating $\int_{x_i}^{x_{i+1}} f(x)dx$. The *rectangle rule* makes the simplest assumption possible, namely that the area under $f(x)$ from x_i to x_{i+1} can be approximated by the area of a rectangle, with width h (the distance from x_i to x_{i+1}) and height given by the value of $f(x)$ either at x_i or at x_{i+1}. An equivalent way of seeing this is that the rectangle rule approximates $f(x)$ as a *constant* from x_i to x_{i+1}, namely a straight (horizontal) line: this means that instead of evaluating the area under the curve it evaluates the area under that straight line.

Let's provide a formula for this before we elaborate further. The analytical expression for the *one-panel version of the rectangle rule* is simply:

$$\int_{x_i}^{x_{i+1}} f(x)dx \approx hf(x_i) \tag{7.10}$$

As noted, we've taken the distance from x_i to x_{i+1} to be fixed, given by Eq. (7.9).

Implicit both in the figure and in the formula above is an assumption, which is easier to elucidate in a specific case. Take the figure, which illustrates the case of five points (and therefore four panels), namely $n = 5$, and focus on any one of the rectangles. Observe we have determined the height of the rectangle as the value of $f(x)$ at the left abscissa, namely $f(x_i)$. This can be referred to as the *left-hand rectangle rule*. We could just as easily have taken the height of the rectangle as the value of $f(x)$ at the right abscissa, namely $f(x_{i+1})$, giving rise to the right-hand rectangle rule.

We now turn to the approximation for the total integral from a to b, namely $\int_a^b f(x)dx$. This is nothing other than the sum of all the one-panel rectangle areas, giving rise to the *composite version of the rectangle rule*:

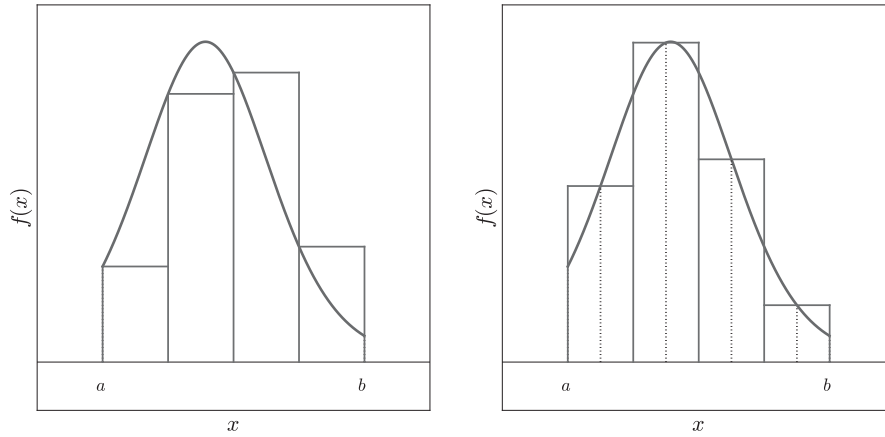


Fig. 7.2 Composite version of rectangle rule (left) and midpoint rule (right)

$$\int_a^b f(x)dx = \sum_{i=0}^{n-2} \int_{x_i}^{x_{i+1}} f(x)dx$$
$$\approx hf(x_0) + hf(x_1) + hf(x_2) + \cdots + hf(x_{n-2}) \tag{7.11}$$

The first line is simply taking the sum and the second line has plugged in the one-panel expression from Eq. (7.10). We can translate this result into the language of weights c_i from Eq. (7.7). The answer is $c_i = h\{1, 1, \ldots, 1, 0\}$, namely, the weights are all equal to h, except for the endpoint at b, where the weight is zero: the rectangle formula is half-open, as it doesn't include one of the two endpoints. [2]

The result in the second line of Eq. (7.11) brings to mind the definition of the Riemann integral: the difference is that here we are taking h as fixed by n, i.e., we are not taking the limit of $h \to 0$ (even if we later try out a larger n, we will never take the actual limit). This is similar to what we saw in section 3.3 on derivatives: the simplest finite-difference formula is just the definition of the derivative, without the limit. One main difference, however, is that in that case making the h smaller led to problems (due to catastrophic cancellation), whereas this won't be an issue for integration.

Error Analysis

We start from a discussion of the approximation error in the one-panel version of the rectangle formula, Eq. (7.10). Take the Taylor expansion of $f(x)$ around x_i:

$$f(x) = f(x_i) + (x - x_i)f'(x_i) + \cdots \tag{7.12}$$

[2] Note that the sum in Eq. (7.7) goes from 0 to $n - 1$ whereas the sum in Eq. (7.11) goes from 0 to $n - 2$.

If we stop at first order, we can rewrite this expression as:

$$f(x) = f(x_i) + (x - x_i)f'(\xi_i) \tag{7.13}$$

where, as usual, ξ_i is a point between x_i and x.

To belabor the obvious: in the previous section, when introducing the rectangle rule, we were dealing with a constant approximation to $f(x)$, *not* with a Taylor series. Thus, in that section we took it for granted that $f(x)$ could be approximated as being a constant from x_i to x_{i+1}, whereas now we are Taylor expanding the actual function $f(x)$, which in general is much more complicated than a simple constant (this is why in what follows we carry around a first derivative, which in general is non-zero).

We now integrate this Taylor series from x_i to x_{i+1} to find:

$$\int_{x_i}^{x_{i+1}} f(x)dx = \int_{x_i}^{x_{i+1}} dx\,[f(x_i) + (x - x_i)f'(\xi_i)] \tag{7.14}$$

where the last term will turn out to be the error term for us. This integral is easy enough that it can be evaluated by hand, but in the interest of establishing the notation for coming sections, we define a helper variable:

$$u = \frac{x - x_i}{h} \tag{7.15}$$

Expressed in terms of u, the integral from x_i to x_{i+1} becomes:

$$\int_{x_i}^{x_{i+1}} f(x)dx = h\int_0^1 du\,[f(x_i) + huf'(\xi_i)] = hf(x_i) + \frac{1}{2}h^2 f'(\xi_i) \tag{7.16}$$

In the second equality we evaluated the integral over u. Comparing this result with the one-panel version of the rectangle formula, Eq. (7.10), we find that the *absolute error in the one-panel rectangle formula* is:

$$\mathcal{E}_i = \frac{1}{2}h^2 f'(\xi_i) \tag{7.17}$$

where we introduced the notation $\mathcal{E}_i$ for the error incurred by approximating one panel.[3]

We now turn to a discussion of the approximation error in the composite version of the rectangle formula, Eq. (7.11). Just like in the first line of Eq. (7.11), we will evaluate the absolute error for the full interval by summing up all the subinterval contributions:

$$\begin{aligned}\mathcal{E} &= \sum_{i=0}^{n-2} \mathcal{E}_i = \frac{1}{2}h^2\left(f'(\xi_0) + f'(\xi_1) + f'(\xi_2) + \cdots + f'(\xi_{n-2})\right) \\ &= \frac{n-1}{2}h^2\left(\frac{f'(\xi_0) + f'(\xi_1) + f'(\xi_2) + \cdots + f'(\xi_{n-2})}{n-1}\right) = \frac{n-1}{2}h^2 f'(\xi)\end{aligned} \tag{7.18}$$

If you're uncomfortable with the first equality, you could instead take Eq. (7.16) and sum all

[3] Note that in chapter 2 we defined the absolute error as "approximate minus exact", $\Delta x = \tilde{x} - x$, see Eq. (2.5). Our definition of the absolute error in this chapter conforms to standard practice and differs by an overall minus sign, which is why we're using a different symbol, $\mathcal{E}$, to denote it.

the panels separately: this will clearly show that the subinterval error contributions simply add up. In the second equality we plugged in the result for $\mathcal{E}_i$ from Eq. (7.17). In the third equality we multiplied and divided with $n-1$, which is the number of terms in the sum. In the fourth equality we identified the term in the parentheses as the arithmetic mean and remembered from elementary calculus that there exists a ξ (from a to b) for which $f'(\xi)$ is equal to the arithmetic mean of f'. Combining the last result with our definition of h in Eq. (7.9), recast as $(n-1)h = b-a$, we find our final result for the *absolute error in the composite rectangle formula*:

$$\mathcal{E} = \frac{b-a}{2} h f'(\xi) \tag{7.19}$$

A point that will re-emerge below: it is wrong to infer from Eq. (7.19) that $\mathcal{E} = ch$ where c is a constant, since $f'(\xi)$ is actually not independent of h (this will become crystal clear in section 7.2.4). If we *do* assume that c is a constant, then we will draw the conclusion that the error in the composite rule is $O(h)$: this is actually correct, if you interpret it as giving the *leading error*. Another pattern that will re-emerge below: the leading error in the composite rule ($O(h)$ here) is worse by one degree compared to the elementary-interval error ($O(h^2)$ here); we say "worse" because for small h we have $h^2 < h$.

7.2.2 Midpoint Rule

We now briefly turn to an improved rule, which is surprisingly similar to the rectangle rule. As expected, we start with the one-panel version; it may help to keep the right panel of Fig. 7.2 in mind. The *midpoint rule* makes the same assumption as the rectangle rule, namely that the area under $f(x)$ from x_i to x_{i+1} can be approximated by the area of a rectangle, with width h (the distance from x_i to x_{i+1}) and height given by the value of $f(x)$. The only difference is that the midpoint rule uses the value of $f(x)$ not at the left or at the right, but at the *midpoint* of the panel. In other words, just like for the rectangle rule, the midpoint rule approximates $f(x)$ as a *constant* from x_i to x_{i+1}, namely a straight (horizontal) line: instead of evaluating the area under the curve it evaluates the area under that straight line. Again, not a great assumption but, as we'll soon see, one that works much better than the rectangle rule. The analytical expression for the *one-panel version of the midpoint rule* is:

$$\int_{x_i}^{x_{i+1}} f(x)dx \approx h f\left(x_i + \frac{h}{2}\right) \tag{7.20}$$

Thinking about the composite version of the midpoint rule, we realize there's a problem here: we would need $f(x)$ evaluated at $x_i + h/2$; if all we have is a table of the form $(x_i, f(x_i))$ for $i = 0, 1, \ldots, n-1$, then we simply don't have access to those function values. This is another way of saying that we cannot really translate our result into the language of

weights c_i from Eq. (7.7), since here we are not evaluating the function at the grid points. Incidentally, you should have realized by now that the midpoint formula is open, since it doesn't include either of the two endpoints. A problem guides you toward the following approximation error in the one-panel version of the midpoint formula:

$$\mathcal{E}_i = \frac{1}{24} h^3 f''(\xi_i) \tag{7.21}$$

Similarly, adding up all the one-panel errors gives us the final result for the *absolute error in the composite midpoint formula*:

$$\mathcal{E} = \frac{b-a}{24} h^2 f''(\xi) \tag{7.22}$$

7.2.3 Integration from Interpolation

Both the rectangle rule and the midpoint rule approximated the area under $f(x)$ from x_i to x_{i+1} in the simplest way possible, namely by the area of a rectangle. The natural next step is to assume that the function is *not* approximated by a constant, i.e., a horizontal line, from x_i to x_{i+1}, but by a straight line, a quadratic, a cubic, and so on. In other words, we see the problem of *interpolation*, studied extensively in chapter 6, emerge here.

It should be straightforward to see that one panel, made up of two consecutive abscissas, is enough to define a general straight line (i.e., not necessarily a flat, horizontal line). Similarly, two panels, made up of three consecutive abscissas, can "anchor" a quadratic, and so on. This is precisely the problem Lagrange interpolation solved, see section 6.2.2. In the language of the problem we are now faced with, we have as input a table of q data points $(x_{i+j}, f(x_{i+j}))$ for $j = 0, 1, \ldots, q-1$ and wish to find the interpolating polynomial that goes through them. For $q = 2$ we get a straight line, for $q = 3$ a quadratic, and so on. Make sure you keep the notation straight: $q = 3$ leads to the three abscissas x_i, x_{i+1}, and x_{i+2}.

Thus, for a given approach the *elementary interval* will depend on the value of q: for $q = 2$ the elementary interval has a width of one panel, for $q = 3$ of two panels, for $q = 4$ of three panels, and so on.[4] More generally, for the case of q points in the elementary interval we wish to approximate the integral:

$$\int_{x_i}^{x_{i+q-1}} f(x)dx \tag{7.23}$$

The way we do this in general is to employ an interpolating polynomial as per Eq. (6.22):

$$p(x) = \sum_{j=0}^{q-1} f(x_{i+j}) L_{i+j}(x) \tag{7.24}$$

[4] We use q for an elementary interval and n, as before, for the composite case, i.e., for the full interval.

Since our nodes go from x_i to x_{i+q-1}, the cardinal polynomials $L_{i+j}(x)$ take the form:

$$L_{i+j}(x) = \frac{\prod_{k=0,k\neq j}^{q-1}(x - x_{i+k})}{\prod_{k=0,k\neq j}^{q-1}(x_{i+j} - x_{i+k})}, \qquad j = 0, 1, \ldots, q-1 \tag{7.25}$$

which is simply a translation of Eq. (6.19) into our present notation. Once again, make sure to disentangle the notation: for $q = 4$ we have $L_i(x)$, $L_{i+1}(x)$, $L_{i+2}(x)$, $L_{i+3}(x)$, each of which is a cubic polynomial. The points they are interpolating over are x_i, x_{i+1}, x_{i+2}, and x_{i+3}.

Using $p(x)$, Newton–Cotes methods in an elementary interval are cast as:

$$\int_{x_i}^{x_{i+q-1}} f(x)dx \approx \int_{x_i}^{x_{i+q-1}} p(x)dx = \sum_{j=0}^{q-1}\left(f(x_{i+j})\int_{x_i}^{x_{i+q-1}} L_{i+j}(x)dx\right) = \sum_{j=0}^{q-1} w_{i+j} f(x_{i+j}) \tag{7.26}$$

We plugged in Eq. (7.24) and then defined the *weights for the elementary interval*:

$$w_{i+j} = \int_{x_i}^{x_{i+q-1}} L_{i+j}(x)dx \tag{7.27}$$

The crucial point is that these weights depend *only* on the cardinal polynomials, *not* on the function $f(x)$ that is being integrated. Thus, for a given q, implying an elementary interval with a width of $q-1$ panels, these weights can be evaluated once and for all, and employed to integrate any function you wish, after the fact.

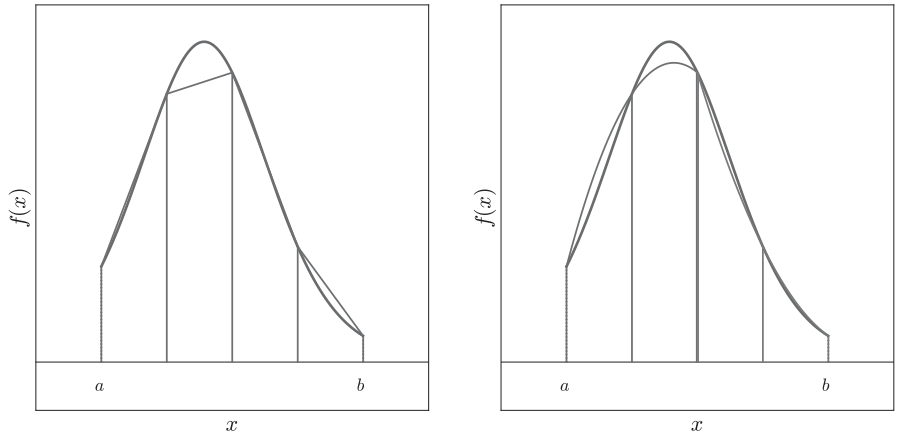
Before seeing the details worked out, let us make a general comment: by focusing on an elementary interval and employing a low-degree polynomial in it, in essence what we're doing in Eq. (7.26) is *piecewise polynomial interpolation and then integration of each interpolant*. This is completely analogous to what we did in section 6.3. Interpolation on an equally spaced grid suffers from serious issues, but focusing on a few points at a time and integrating in elementary intervals mostly gets rid of these problems.

7.2.4 Trapezoid Rule

It may help to look at the left panel of Fig. 7.3 in what follows, but keep in mind that it is showing the composite version of the trapezoid rule, whereas we will, as before, first discuss our new rule in its elementary-interval version.

The previous discussion may be too abstract, so let us make it tangible. We start from the case of $q = 2$, giving rise to what is known as the *trapezoid rule*.[5] We have the two

[5] Strangely enough, this is typically called the *trapezoidal* rule. But we don't say *rectangular* rule, do we? More importantly, the rule itself is most certainly *not* shaped like a trapezoid. How could it? It's a rule. Note that we're not being pedantic here. A purist would point out that Proclus introduced the term *trapezoid* to denote a quadrilateral figure no two of whose sides are parallel. The same purist would insist that this be called the *trapezium* rule and would cringe were anyone to refer to it as the *trapeziform* rule.



Composite version of trapezoid rule (left) and Simpson's rule (right) **Fig. 7.3**

points $(x_i, f(x_i))$ and $(x_{i+1}, f(x_{i+1}))$ and the two cardinal polynomials from Eq. (7.25):

$$L_i(x) = \frac{x - x_{i+1}}{x_i - x_{i+1}} = -\frac{x - x_{i+1}}{h}, \qquad L_{i+1}(x) = \frac{x - x_i}{x_{i+1} - x_i} = \frac{x - x_i}{h} \tag{7.28}$$

We can immediately evaluate the two elementary weights from Eq. (7.27):

$$\begin{aligned} w_i &= \int_{x_i}^{x_{i+1}} L_i(x)dx = -\frac{1}{h}\int_{x_i}^{x_{i+1}} (x - x_{i+1})\,dx = \frac{h}{2} \\ w_{i+1} &= \int_{x_i}^{x_{i+1}} L_{i+1}(x)dx = \frac{1}{h}\int_{x_i}^{x_{i+1}} (x - x_i)\,dx = \frac{h}{2} = w_i \end{aligned} \tag{7.29}$$

where we used the fact that $x_{i+1} = x_i + h$. It's worth noting at this point that the sum of weights in the one-panel version of the trapezoid rule is h ($h/2$ for the point at x_i and $h/2$ for the point at x_{i+1}). This is actually a general result, which you will show in a problem: *for any Newton–Cotes integration rule, the sum of all the weights is equal to the width of the elementary interval*. Here the elementary interval is simply one panel, so the sum of the weights is h. We will see more examples of this below.

Taking these elementary weights and plugging them into Eq. (7.26) leads to the *one-panel version of the trapezoid rule*:

$$\int_{x_i}^{x_{i+1}} f(x)dx \approx \frac{h}{2}[f(x_i) + f(x_{i+1})] \tag{7.30}$$

which merely reiterates the fact that each of the two points making up our elementary interval gets the same weight, $h/2$.

We now turn to the approximation for the total integral from a to b, namely $\int_a^b f(x)dx$. This is nothing other than the sum of all the one-panel trapezoid areas, giving rise to the *composite version of the trapezoid rule*:

$$\int_a^b f(x)dx = \sum_{i=0}^{n-2} \int_{x_i}^{x_{i+1}} f(x)dx$$
$$\approx \frac{h}{2}f(x_0) + hf(x_1) + hf(x_2) + \cdots + hf(x_{n-3}) + hf(x_{n-2}) + \frac{h}{2}f(x_{n-1}) \tag{7.31}$$

The first line is simply taking the sum; the second line has plugged in the one-panel expression from Eq. (7.30) and grouped terms together: each intermediate term is counted twice, so the 2 in the denominator cancels there. It is now trivial to translate this result into the language of weights c_i from Eq. (7.7). The answer is simply:

$$c_i = h\left\{\frac{1}{2}, 1, \ldots, 1, \frac{1}{2}\right\} \tag{7.32}$$

namely, the weights are all equal to h, except for the endpoints, where the weight is $h/2$; the trapezoid formula is closed, meaning that it includes both endpoints.[6] Note, finally, that we have been careful to use different symbols for the weights in an elementary interval and for the overall Newton–Cotes weights: the former are denoted by w_{i+j} and the latter by c_i.

Just to avoid confusion, we observe that the trapezoid rule may approximate a function by a sequence of straight lines, but that's not something you need to worry about when using the rule: Eq. (7.31) *is* the composite trapezoid rule so you could, if you want, forget about where it came from. (Remember: the composite rule is made up of many trapezoids, whereas the one-panel version, of a single trapezoid.)

Error Analysis

In a problem, you are asked to carry out an error analysis for the trapezoid rule by generalizing what we did for the rectangle rule in an earlier section. However, since we just introduced the trapezoid rule using the Lagrange-interpolation machinery, it makes sense to turn to the latter for guidance on the error behavior.

As you may recall, in section 6.2.3 we derived a general error formula for polynomial interpolation, Eq. (6.38). For the present case of $q = 2$ this takes the form:

$$f(x) - p(x) = \frac{1}{2!}f''(\xi_i)(x - x_i)(x - x_{i+1}) \tag{7.33}$$

Integrating this difference from x_i to x_{i+1} will then give us the error in the one-panel version of the trapezoid rule. We have:

$$\mathcal{E}_i = \int_{x_i}^{x_{i+1}} \left(\frac{1}{2}f''(\xi_i)(x - x_i)(x - x_{i+1})\right)dx = \frac{1}{2}f''(\xi_i)\int_{x_i}^{x_{i+1}} (x - x_i)(x - x_{i+1})dx \tag{7.34}$$

This might seem like an improper thing to do: after all, the ξ in Eq. (6.38) implicitly

[6] Note that the sum in Eq. (7.7) goes from 0 to $n - 1$ whereas the sum in the first line of Eq. (7.31) goes from 0 to $n - 2$, but each term in that sum includes both $f(x_i)$ and $f(x_{i+1})$, as per Eq. (7.30).

depended on the placement of the nodes. However, you can use the extended version of the integral mean-value theorem: this is justified because $(x - x_i)(x - x_{i+1})$ has a constant sign in our interval. If you then do the integral in the last step and, once again, employ the fact that $x_{i+1} = x_i + h$, you find the following *absolute error in the one-panel trapezoid formula*:

$$\mathcal{E}_i = -\frac{1}{12}h^3 f''(\xi_i) \tag{7.35}$$

Thus, the leading term in the error for the one-panel case is $O(h^3)$.

We now turn to a discussion of the approximation error in the composite version of the trapezoid formula, Eq. (7.31). Just like in the first line of Eq. (7.31), we will evaluate the absolute error for the full interval by summing up all the subinterval contributions:

$$\begin{aligned}\mathcal{E} = \sum_{i=0}^{n-2} \mathcal{E}_i &= -\frac{1}{12}h^3 \left(f''(\xi_0) + f''(\xi_1) + f''(\xi_2) + \cdots + f''(\xi_{n-2})\right) \\ &= -\frac{n-1}{12}h^3 \left(\frac{f''(\xi_0) + f''(\xi_1) + f''(\xi_2) + \cdots + f''(\xi_{n-2})}{n-1}\right) = -\frac{n-1}{12}h^3 f''(\xi)\end{aligned} \tag{7.36}$$

In the second equality we simply plugged in the result for $\mathcal{E}_i$ from Eq. (7.35). In the third equality we multiplied and divided with $n - 1$, which is the number of terms in the sum. In the fourth equality we identified a ξ (from a to b) for which $f''(\xi)$ is equal to the arithmetic mean of f''. Combining the last line with our definition of h in Eq. (7.9), we find our final result for the *absolute error in the composite trapezoid formula*:

$$\mathcal{E} = -\frac{b-a}{12}h^2 f''(\xi) \tag{7.37}$$

Observe that this error contains a second derivative, so *for polynomials of up to first degree, the composite trapezoid rule is exact*. Since we derived the trapezoid rule by approximating $f(x)$ by straight lines, this isn't really surprising. This approximation error, $O(h^2)$, is of the same order as the approximation error of the midpoint rule, Eq. (7.22). As a matter of fact, in absolute value, the error in the midpoint rule, Eq. (7.22), is smaller by a factor of 2 than the error in the trapezoid rule, Eq. (7.37). This is quite interesting: the midpoint rule uses a cruder shape (a rectangle) than the trapezoid one, but manages to be more accurate, by using the midpoint instead of the endpoints of each panel. Of course, that is at the price of not employing the function values at the grid points, $f(x_i)$.

Beyond the Leading Error

It is wrong to infer from Eq. (7.37) that $\mathcal{E} = ch^2$ where c is a constant, since $f''(\xi)$ is actually not independent of h. The question then arises if we can do better, i.e., if we can quantify the error to higher orders. This will turn out to be a very fruitful avenue: section 7.4

will essentially rely on the behavior of the trapezoid error beyond the leading term. Thus, it is important to investigate the trapezoid-rule error in more detail.

Help comes from a (somewhat) unexpected place: in a problem in chapter 2 we introduced Bernoulli numbers and the *Euler–Maclaurin summation formula*, Eq. (2.120):[7]

$$\int_0^1 g(t)dt = \frac{g(0)}{2} + \frac{g(1)}{2} - \sum_{j=1}^{m} \frac{1}{(2j)!} B_{2j} \left[g^{(2j-1)}(1) - g^{(2j-1)}(0) \right] + R \tag{7.38}$$

where the B_{2j}'s are Bernoulli numbers and the R is the higher-order error term, given in terms of an integral over a Bernoulli polynomial. We wish to use this equation to get guidance on the behavior of the trapezoid rule. With that in mind, we make the transformation $x = x_i + ht$ and take $g(t) = f(x) = f(x_i + ht)$. From this definition, together with the chain rule, we find for the first derivative $g'(t) = f'(x)h$, and similarly $g^{(k)}(t) = f^{(k)}(x)h^k$ for higher derivatives. Putting everything together, the one-panel integral of $f(x)$ becomes:

$$\int_{x_i}^{x_{i+1}} f(x)dx = h \int_0^1 g(t)dt = \frac{h}{2}(f(x_i)+f(x_{i+1})) - \sum_{j=1}^{m} \frac{1}{(2j)!} B_{2j}[f^{(2j-1)}(x_{i+1}) - f^{(2j-1)}(x_i)]h^{2j} \tag{7.39}$$

where we dropped the R term, since we're only interested in establishing the order pattern beyond the leading term. The first term on the right-hand side already matches the one-panel version of the trapezoid rule, Eq. (7.30). This means that we've already computed the one-panel error, $\mathcal{E}_i$, beyond leading order.

Instead of focusing on the one-panel result, we will press on: adding together all the one-panel expressions, as in Eq. (7.31) above, we find:

$$\begin{aligned} \int_a^b f(x)dx = {} & \frac{h}{2} f(x_0) + hf(x_1) + hf(x_2) + \cdots + hf(x_{n-3}) + hf(x_{n-2}) + \frac{h}{2} f(x_{n-1}) \\ & - \sum_{j=1}^{m} \frac{1}{(2j)!} B_{2j} \left[f^{(2j-1)}(b) - f^{(2j-1)}(a) \right] h^{2j} \end{aligned} \tag{7.40}$$

where on the first line we identify the trapezoid-rule approximation to the integral; on the second line we are happy to see the "intermediate" series all cancel; this is a version of a telescoping series,[8] only this time each term that cancelled was a sum over j. In the end, we're left with a single sum over derivative values at the *endpoints* of our integral, only.

Plugging in the values of the first few Bernoulli numbers, which you computed in the chapter 2 problem, the *absolute error in the trapezoid rule to all orders* takes the form:

$$\begin{aligned} \mathcal{E} = {} & -\frac{h^2}{12} \left[f'(b) - f'(a) \right] + \frac{h^4}{720} \left[f'''(b) - f'''(a) \right] \\ & - \frac{h^6}{30\,240} \left[f^{(5)}(b) - f^{(5)}(a) \right] + \frac{h^8}{1\,209\,600} \left[f^{(7)}(b) - f^{(7)}(a) \right] + \cdots \end{aligned} \tag{7.41}$$

[7] If you didn't solve that problem, and therefore would prefer not to rely on it, you can solve a problem in the present chapter, which guides you toward proving our main result in Eq. (7.41) "by hand".

[8] We encountered another telescoping series in a problem in chapter 4.

This is a significant formula, which tells us several things. We see, first, that the error of the trapezoid rule contains only *even* powers in h and, second, that the coefficient of each h^{2j} term depends only on a *difference* of derivative values at the *endpoints*. The latter point means that the error is not impacted by the behavior of the derivatives at intermediate points. We will return to the first point when we introduce Romberg integration in section 7.4 and to the second point in our discussion of analytical features in section 7.6.

7.2.5 Simpson's Rule

As you may have guessed, *Simpson's rule* is the natural continuation of the Lagrange-interpolation process we saw above; it uses $q = 3$ in an elementary interval, namely the three abscissas x_i, x_{i+1}, and x_{i+2}. The trapezoid rule fit a straight line through two points, so Simpson's rule fits a quadratic through three points (making up two panels); in a problem, you are asked to employ "naive", i.e., monomial, interpolation, but here we will capitalize on the infrastructure developed in chapter 6. It may help to look at the right panel of Fig. 7.3 from here onward; this shows the case of four panels, i.e., two elementary intervals each of which has width $2h$.

We are dealing with three points $(x_i, f(x_i))$, $(x_{i+1}, f(x_{i+1}))$, and $(x_{i+2}, f(x_{i+2}))$, and three cardinal polynomials from Eq. (7.25):

$$\begin{aligned} L_i(x) &= \frac{(x - x_{i+1})(x - x_{i+2})}{(x_i - x_{i+1})(x_i - x_{i+2})} \\ L_{i+1}(x) &= \frac{(x - x_i)(x - x_{i+2})}{(x_{i+1} - x_i)(x_{i+1} - x_{i+2})} \\ L_{i+2}(x) &= \frac{(x - x_i)(x - x_{i+1})}{(x_{i+2} - x_i)(x_{i+2} - x_{i+1})} \end{aligned} \tag{7.42}$$

Each of these is a quadratic polynomial. We would now like to compute the three elementary weights; as you can see from Eq. (7.27), this will require integrating from x_i to x_{i+2}, i.e., in an elementary interval. In order to streamline the evaluation of these integrals, we define a helper variable:[9]

$$u = \frac{x - x_{i+1}}{h} \tag{7.43}$$

and notice that the denominators in Eq. (7.42) are simply $-h$, $-2h$, h, and so on. Then, the elementary weights from Eq. (7.27) are:

$$\begin{aligned} w_i &= \int_{x_i}^{x_{i+2}} L_i(x)dx = \frac{h}{2}\int_{-1}^{1} du\, u(u-1) = \frac{h}{3} \\ w_{i+1} &= \int_{x_i}^{x_{i+2}} L_{i+1}(x)dx = -h\int_{-1}^{1} du\, (u+1)(u-1) = \frac{4h}{3} \\ w_{i+2} &= \int_{x_i}^{x_{i+2}} L_{i+2}(x)dx = \frac{h}{2}\int_{-1}^{1} du\, (u+1)u = \frac{h}{3} \end{aligned} \tag{7.44}$$

[9] This is similar to what we did when deriving the leading error in the rectangle rule, Eq. (7.15), only this time we employ the midpoint of our elementary interval in the numerator, x_{i+1}.

Taking these elementary weights and plugging them into Eq. (7.26) leads to the *two-panel version of Simpson's rule*:

$$\int_{x_i}^{x_{i+2}} f(x)dx \approx \frac{h}{3}\left[f(x_i) + 4f(x_{i+1}) + f(x_{i+2})\right] \tag{7.45}$$

As we observed when introducing the one-panel version of the trapezoid rule in Eq. (7.30), we notice that the sum of all the weights in this elementary interval is $(1 + 4 + 1)h/3 = 2h$, consistent with the fact that the elementary interval in this case has width $2h$.

We now turn to the approximation for the total integral from a to b, namely $\int_a^b f(x)dx$. This is nothing other than the sum of all the two-panel elementary areas, giving rise to the *composite version of Simpson's rule*:

$$\begin{aligned}\int_a^b f(x)dx &= \sum_{i=0,2,4,\ldots}^{n-3} \int_{x_i}^{x_{i+2}} f(x)dx \\ &\approx \frac{h}{3}f(x_0) + \frac{4h}{3}f(x_1) + \frac{2h}{3}f(x_2) + \cdots + \frac{2h}{3}f(x_{n-3}) + \frac{4h}{3}f(x_{n-2}) + \frac{h}{3}f(x_{n-1})\end{aligned} \tag{7.46}$$

The first equality is simply taking the sum, but this time we're careful since we're moving in steps of $2h$: this is why the sum over i goes in steps of 2 and ends at $n - 3$ (so the last x_{i+2} can be x_{n-1}, which is our last point). The second equality has plugged in the two-panel expression from Eq. (7.45) and grouped terms together: each intermediate term is counted twice, so we go from $h/3$ to $2h/3$ for those points. It is now trivial to translate this result into the language of weights c_i from Eq. (7.7). The answer is simply:

$$c_i = \frac{h}{3}\{1, 4, 2, 4 \ldots, 2, 4, 1\} \tag{7.47}$$

namely, the weights for the two endpoints are $h/3$, for each "middle" point the weight is $4h/3$, whereas the intermediate/matching points get a weight of $2h/3$ as already mentioned. Obviously, Simpson's formula is closed, as it includes both endpoints.

Here's a very important point that was implicit up to here: to use Simpson's rule, you have to use an even number of panels. This is because the "elementary interval" version gives us the two-panel answer: $\int_{x_i}^{x_{i+2}} f(x)dx$. We actually took this fact for granted in the right panel of Fig. 7.3, which shows the case of $n = 5$ points and therefore four panels. We used $n = 5$ in the earlier methods, too, but there this was by choice, since those methods could use an even or an odd number of panels and work just fine. To summarize this significant point, *Simpson's rule requires an even number of panels and therefore an odd number of points n*.

Error Analysis

The natural next step would be to generalize our error analysis for the trapezoid rule, leading up to Eq. (7.35), for the case of Simpson's rule. You are guided toward such an analysis using the general interpolation error formula, Eq. (6.38), in one of the problems. Here we will follow a different route: we will carry out the error analysis for Simpson's rule by employing a Taylor-series approach, similarly to what we did for the rectangle rule starting in Eq. (7.12). Since Simpson's rule is a more accurate method, we'll have to keep more terms in the Taylor series expansion this time.

As usual, we start from the elementary interval, namely from the two-panel version of Simpson's rule in Eq. (7.45). Taylor expanding $f(x)$ around x_{i+1} gives us:

$$f(x) = f(x_{i+1}) + (x - x_{i+1})f'(x_{i+1}) + \frac{1}{2}(x - x_{i+1})^2 f''(x_{i+1}) + \frac{1}{6}(x - x_{i+1})^3 f'''(x_{i+1}) + \frac{1}{24}(x - x_{i+1})^4 f^{(4)}(\xi_{i+1}) \tag{7.48}$$

where we have stopped at fourth order; as usual, ξ_{i+1} is a point between x_{i+1} and x (we're calling it ξ_{i+1}, instead of ξ_i, for purely aesthetic reasons). We wish to integrate this Taylor series from x_i to x_{i+2}: to make this calculation easier, we employ u, the helper variable from Eq. (7.43). Expressed in terms of u, the integral from x_i to x_{i+2} becomes:

$$\begin{aligned}\int_{x_i}^{x_{i+2}} f(x)dx &= h\int_{-1}^{1} du\Big[f(x_{i+1}) + huf'(x_{i+1}) + \frac{1}{2}h^2u^2f''(x_{i+1}) \\ &\qquad + \frac{1}{6}h^3u^3f'''(x_{i+1}) + \frac{1}{24}h^4u^4f^{(4)}(\xi_{i+1})\Big] \\ &= 2hf(x_{i+1}) + \frac{1}{3}h^3f''(x_{i+1}) + \frac{1}{60}h^5f^{(4)}(\xi_{i+1})\end{aligned} \tag{7.49}$$

where, crucially, the third-derivative term cancelled. We're not done yet. The first term on the last line of our result contains $2hf(x_i)$, which looks different from the two-panel version of Simpson's formula, Eq. (7.45). In order to make further progress, we will plug in an approximation for the second derivative term $f''(x_{i+1})$. We get this from Eq. (3.39):

$$f''(x_{i+1}) = \frac{f(x_i) + f(x_{i+2}) - 2f(x_{i+1})}{h^2} - \frac{h^2}{12}f^{(4)}(\xi_{i+1}) \tag{7.50}$$

where we assumed $x = x_{i+1}$ and took the liberty of writing $f(x_{i+1})$ instead of $f(x_i + h)$ (and $f(x_{i+2})$ instead of $f(x_i + 2h)$). We also dropped all higher-order terms (at the cost of using ξ_{i+1} in the last term): the actual error term came from Eq. (3.31) with $h \to 2h$.[10]

Now, plugging our result for $f''(x_{i+1})$ from Eq. (7.50) into our expression for the two-panel integral, Eq. (7.49), we find:

$$\int_{x_i}^{x_{i+2}} f(x)dx = 2hf(x_{i+1}) + \frac{1}{3}h^3\left[\frac{f(x_i) + f(x_{i+2}) - 2f(x_{i+1})}{h^2} - \frac{h^2}{12}f^{(4)}(\xi_{i+1})\right] + \frac{1}{60}h^5f^{(4)}(\xi_{i+1})$$

[10] Strictly speaking, this ξ_{i+1} is different from the one in the Taylor expansion Eq. (7.49). However, even if we used a different symbol for it now, the end result would be the same, so we don't bother.

$$= \frac{h}{3}\left[f(x_i) + 4f(x_{i+1}) + f(x_{i+2})\right] - \frac{1}{90}h^5 f^{(4)}(\xi_{i+1}) \tag{7.51}$$

If you're troubled by this, you should observe that the next term in the finite-difference expansion would be proportional to h^4 which, together with the h^3 outside the brackets would give a h^7; we drop this, since it's of higher order than our leading error.

We're now happy to see that this result loooks exactly like the two-panel version of Simpson's formula, Eq. (7.45), with the addition of the *absolute error in the two-panel Simpson's formula*:

$$\mathcal{E}_i = -\frac{1}{90}h^5 f^{(4)}(\xi_{i+1}) \tag{7.52}$$

Thus, the leading term in the two-panel error is $O(h^5)$.

We now turn to a discussion of the approximation error in the composite version of Simpson's formula, Eq. (7.46). Just like in the first line of Eq. (7.46), we will evaluate the absolute error for the full interval by summing up all the elementary contributions:

$$\begin{aligned}\mathcal{E} &= \sum_{i=0,2,4,\ldots}^{n-3} \mathcal{E}_i = -\frac{1}{90}h^5\left(f^{(4)}(\xi_1) + f^{(4)}(\xi_3) + f^{(4)}(\xi_5) + \cdots + f^{(4)}(\xi_{n-2})\right) \\ &= -\frac{(n-1)/2}{90}h^5\left(\frac{f^{(4)}(\xi_1) + f^{(4)}(\xi_3) + f^{(4)}(\xi_5) + \cdots + f^{(4)}(\xi_{n-2})}{(n-1)/2}\right) = -\frac{n-1}{180}h^5 f^{(4)}(\xi)\end{aligned} \tag{7.53}$$

In the second equality we plugged in the result for $\mathcal{E}_i$ from Eq. (7.52). In the third equality we multiplied and divided with $(n-1)/2$, which is the number of terms in the sum. In the fourth equality we employed a ξ (from a to b) for which $f^{(4)}(\xi)$ is equal to the arithmetic mean of $f^{(4)}$. Combining the last expression with our definition of h, we find our final result for the *absolute error in the composite Simpson's formula*:

$$\mathcal{E} = -\frac{b-a}{180}h^4 f^{(4)}(\xi) \tag{7.54}$$

Observe that this error contains a fourth derivative, so *for polynomials of up to third degree, the composite Simpson's rule is exact*. Given what we know about Simpson's rule, this is surprising: we found that this method is exact for all cubic polynomials, even though we derived it using a quadratic polynomial (remember when the third-derivative term in Eq. (7.49) vanished?). This approximation error, $O(h^4)$, is much better than any of the other quadrature errors we've encountered up to this point (e.g., midpoint or trapezoid). We will offer some advice below on which integration method you should choose in general; for now, we note that if you're not dealing with poorly behaved functions (which may have Taylor series that behave improperly, in the sense of having successive terms grow in magnitude) Simpson's is a good method.

Table 7.1 Features of selected Newton–Cotes methods in an elementary interval

Method	Panels	Polynomial	Weights	Absolute error
Rectangle	1	Constant	h	$\frac{1}{2}h^2 f'(\xi_i)$
Trapezoid	1	Straight line	$\frac{h}{2}(1,1)$	$-\frac{1}{12}h^3 f''(\xi_i)$
Simpson's 1/3	2	Quadratic	$\frac{h}{3}(1,4,1)$	$-\frac{1}{90}h^5 f^{(4)}(\xi_i)$
Simpson's 3/8	3	Cubic	$\frac{3h}{8}(1,3,3,1)$	$-\frac{3}{80}h^5 f^{(4)}(\xi_i)$
Boole	4	Quartic	$\frac{2h}{45}(7,32,12,32,7)$	$-\frac{8}{945}h^7 f^{(6)}(\xi_i)$

7.2.6 Summary of Results

It shouldn't be hard to see how this process continues: we keep raising the power of the polynomial used to approximate $f(x)$ in an elementary interval and hope for the best. Using a cubic polynomial to approximate the function within the elementary area gives rise to *Simpson's 3/8 rule*[11] (which needs three panels) and, similarly, using a quartic polynomial in the four-panel problem leads to *Boole's rule*. These are totally straightforward, so you are asked to implement them in the problem set. As you learned in the previous chapter, this process cannot go on *ad infinitum*: interpolation using high-degree polynomials on an equally spaced grid will eventually misbehave.

What we have so far, however, already allows us to notice a pattern: *a Newton–Cotes method that requires an even number of panels as an elementary interval gives the exact answer for the full interval a to b for polynomials of one degree higher than the number of panels in that elementary interval* (e.g., Simpson's rule needs two panels and is exact for polynomials up to third degree, whereas Boole's rule needs four panels and is exact for polynomials up to fifth degree). On the other hand, Newton–Cotes methods that contain an odd number of panels in an elementary interval are exact for the full interval a to b for polynomials of the same degree as the number of panels in that elementary interval (e.g., the trapezoid rule uses one panel and is exact for polynomials up to first degree, whereas Simpson's 3/8 rule needs three panels and is exact for polynomials up to third degree).[12]

Since we've introduced a fair number of Newton–Cotes formulas, exploring the underlying approximation (e.g., quadratic polynomial), the weights involved, as well as the error behavior, it might be helpful to produce a compendium of the most important results, see Table 7.1. In this table "panels" refers to the number of panels included in an elementary interval; "polynomial" is referring to the order of the polynomial used to approximate the function in each elementary interval; the "weights" here are the coefficients of the function values in an elementary interval; the "absolute error" is $\mathcal{E}_i$, i.e., the error only in one

[11] Simpson's rule from Eq. (7.46) is sometimes called "Simpson's 1/3 rule" because the first weight is 1/3 (times h). As you can imagine, "Simpson's 3/8 rule" employs different weights. Given the popularity of "Simpson's 1/3 rule" it is often called simply "Simpson's rule".

[12] We further observe that all the closed formulas we've seen (trapezoid, the two Simpson's rules, and Boole's rule) have an error coefficient that is negative, whereas the half-open or open methods (rectangle and midpoint rules) have an error coefficient that is positive.

elementary interval; as a result, the powers of h shown are one degree higher than those of the composite rules (e.g., the composite trapezoid error is $O(h^2)$ whereas the elementary trapezoid error listed here is $O(h^3)$).

7.2.7 Implementation

We are now in a position to implement the composite rectangle rule, Eq. (7.11), the composite trapezoid rule, Eq. (7.31), as well as the composite Simpson's rule, Eq. (7.46). To make things tangible, we will evaluate the integral giving the electrostatic potential in Eq. (7.2), where we know the answer analytically. An implementation is given in Code 7.1.

Our grid of abscissas `xs` is the one from Eq. (7.8). The only exception is in the function `rectangle()`, where we don't include x_{n-1} (also known as b) in the sum. Note that if we were really efficiency-oriented, we would have stored neither the x_i nor the $f(x_i)$: we would have simply summed the latter up. Instead, we are here opting for a more reader-friendly solution, as usual. We don't separately multiply each $f(x_i)$ with h, as per Eq. (7.11): instead of having several multiplications, we simply factor the h out and have only one multiplication at the end. We didn't bother storing the weights c_i in a separate array, since they're so straightforward. Turning to the `trapezoid()` function: the main difference here is that we are now explicitly storing the weights c_i in a separate array, in order to avoid any bookkeeping errors (and for ease of reading). The first and last elements of the array have weight $h/2$ and all other ones are h (where we again multiply with h only at the end, to avoid roundoff error creep). You should observe that we are using a `numpy` array for the weights `cs` that contains `n` elements and similarly an array for the grid points `xs` that also contains `n` elements. The implementation of Simpson's rule involves slightly more subtle bookkeeping: we distinguish between odd and even slots (and use different values for the two endpoints at a and at b). Our function does *not* check to see if n is odd, so would give wrong answers if employed for an even value of n.

The output of running this code is:

```
0.88137358702
0.884290734036   0.881361801848   0.881373587255
```

Thus, for $n = 51$ points, the absolute error for: (a) the rectangle rule is in the third digit after the decimal point, (b) the trapezoid rule is in the fifth digit after the decimal point, and (c) Simpson's rule is in the tenth digit after the decimal point. Remember: for 51 points, we have $h = 0.02$ (the error in the rectangle rule being $O(h)$), while $h^2 = 0.0004$: since the error in the trapezoid rule is $O(h^2)$, we have an improvement of two orders of magnitude. Similarly, the error in Simpson's rule is $O(h^4)$; $h^4 = 1.6 \times 10^{-7}$, but there's also a 180 in the denominator of Eq. (7.54).

We now decide to employ this code for many different values of h (or, equivalently, of n). The result is shown in Fig. 7.4. Note that the x axis is showing n on a logarithmic scale, whereas the corresponding figure in the chapter on derivatives, Fig. 3.2, had h on a logarithmic scale. (In other words, here as the grid gets finer we are moving to the right, whereas in the derivatives figure a finer grid meant moving to the left.)

newtoncotes.py **Code 7.1**

```
import numpy as np

def f(x):
     return 1/np.sqrt(x**2 + 1)

def rectangle(f,a,b,n):
     h = (b-a)/(n-1)
     xs = a + np.arange(n-1)*h
     fs = f(xs)
     return h*np.sum(fs)

def trapezoid(f,a,b,n):
     h = (b-a)/(n-1)
     xs = a + np.arange(n)*h
     cs = np.ones(n); cs[0] = 0.5; cs[-1] = 0.5
     contribs = cs*f(xs)
     return h*np.sum(contribs)

def simpson(f,a,b,n):
     h = (b-a)/(n-1)
     xs = a + np.arange(n)*h
     cs = 2*np.ones(n)
     cs[1::2] = 4; cs[0] = 1; cs[-1] = 1
     contribs = cs*f(xs)
     return (h/3)*np.sum(contribs)

if __name__ == '__main__':
     ans = np.log(1 + np.sqrt(2))
     print(ans)

     for integrator in (rectangle, trapezoid, simpson):
          print(integrator(f, 0., 1., 51), end=" ")
```

You should interpret this figure as follows: a decrease in the absolute error (moving down the y axis) means we're doing a good job; moving to the right on the x axis means we're adding more points. Let us start from discussing the rectangle-rule results: we see that this is a pretty slow method, as the absolute error for this formula decreases only very slowly. Even when using a million points, the absolute error is $\approx 10^{-7}$, which is quite large. Even so,

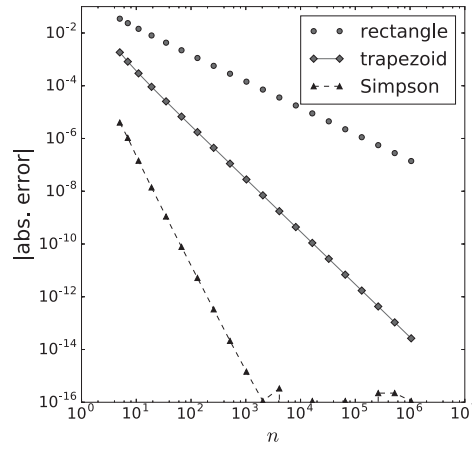


Fig. 7.4 Log-log plot resulting from the rectangle, trapezoid, and Simpson's rules

it's worth observing that the rectangle-rule results fall on a straight line, implying that we are only witnessing the approximation error here, not any roundoff error. In other words, as n gets larger by an order of magnitude, the absolute error gets smaller by an order of magnitude: recall that for this approach the approximation error is $O(h)$. You should keep in mind Eq. (7.9), which tells us that $h = (b - a)/(n - 1)$.

The results for the trapezoid rule are analogous: as n gets larger by an order of magnitude, the absolute error gets smaller by two orders of magnitude: for this approach the approximation error is $O(h^2)$. This process doesn't seem to have reached a minimum: we simply stopped after roughly a million points, where the absolute error is $\approx 10^{-14}$.

The results for Simpson's (1/3) rule are not that strange, either: as n gets larger by an order of magnitude, the absolute error gets smaller by roughly four orders of magnitude: recall that for this approach the approximation error is $O(h^4)$. This process is so fast that after about 1000 points or so we have reached a minimum absolute error of $\approx 10^{-16}$, which is as good as we're going to get. After that, we run up against numerical roundoff issues, so as n is increased beyond that point the absolute error wobbles.[13]

7.3 Adaptive Integration

In the previous section we examined the absolute error by comparing the output of each Python function (corresponding to a quadrature rule) with the analytically known answer for the integral. However, there are going to be (several) situations where you cannot evaluate an integral analytically, so it will be impossible to quantify the absolute error in this manner.

[13] It's worth observing that in Fig. 3.2 the central-difference formula (crudely speaking the analogue to the trapezoid rule) reached a minimum absolute error of $\approx 10^{-10}$, which is four orders of magnitude worse than what we found for the trapezoid rule in Fig. 7.4.

In practice we are often faced with a very different problem: we know that we can only tolerate an answer that is accurate to at least, say, 10 decimal places, and are now wondering which n to pick and whether or not we should trust our answer. One solution to this problem is to keep increasing the number of points n by hand and seeing if the answer appears to be converging toward a given value. This is fine for simple "back of the envelope" calculations, but for dependable answers there should be a way of automating this entire process. This can be done as part of what is called *adaptive integration*: you provide your function, the region where you want it integrated, and your error tolerance; your method then determines the optimal n (without needing to inform you of this) and gives you a value for the integral along with an estimate of how accurate it is.

7.3.1 Doubling the Number of Panels

As usual, our task is to evaluate $\int_a^b f(x)dx$, which we now call I for ease of reference. In the present section, it will be helpful to keep in mind both the number of points n and the number of panels $N = n - 1$ separately. We will now examine elementary consequences of the error analyses for the rectangle, trapezoid, and Simpson's rules that we discussed above. Regardless of which quadrature method we employ, we denote the result for the value of the integral I_N when using N panels. We also modify our notation on the panel width: as per Eq. (7.9), we define:

$$h_N = \frac{b-a}{N} \tag{7.55}$$

The only differences are that we now use the number of panels N in the denominator (instead of using $n - 1$) and that we called the panel width h_N, to ensure that we keep track of which width we're talking about. For example, the estimate for our integral will be called I_{16} for the case of $N = 16$ panels (for which we have $n = N + 1 = 17$ points), making use of panel width h_{16}; similarly, I_{128} for the case of $N = 128$ panels (for which we have $n = N + 1 = 129$ points), making use of panel width h_{128}.

We will now focus on what happens when we start from a given panel number N and then change the panel number to $N' = 2N$. We're simply doubling the panel number, but it will turn out to be convenient for us to use a new variable (N') to denote the new panel number. This is almost too trivial to write out explicitly but, just in case, here's how we can relate the new panel width $h_{N'}$ to the old panel width h_N:

$$h_{N'} = \frac{b-a}{N'} = \frac{b-a}{2N} = \frac{1}{2}\frac{b-a}{N} = \frac{h_N}{2} \tag{7.56}$$

This isn't rocket surgery: doubling the number of panels, you halve the panel width.

We will now see what happens to the error as we go from N panels to N' panels. In essence, we are doing (for a given integration rule) two calculations of the integral, one giving I_N and one giving $I_{N'}$. To keep things manageable, we will only study Simpson's rule, leaving the analogous derivation for the rectangle and trapezoid rules as a problem.

Using our new notation, we know that to leading order in the error, the total absolute error for a Simpson's-rule calculation with N panels is $\mathcal{E}_N = ch_N^4$, as per Eq. (7.54); as above, c actually depends on h_N, but we here take it to be constant. This means that our

estimate for the value of I when employing N panels can be related to its total absolute error as follows:

$$I = I_N + ch_N^4 \tag{7.57}$$

This is nothing other than Eq. (7.54), but this time using our new notation. Similarly, we can apply the same equation to the case of $N' = 2N$ panels.:

$$I = I_{N'} + ch_{N'}^4 \tag{7.58}$$

Equating the right-hand sides of the last two equations gives us:

$$I_N + ch_N^4 = I_{N'} + ch_{N'}^4 \tag{7.59}$$

which can be manipulated to give:

$$I_{N'} - I_N = ch_N^4 - ch_{N'}^4 \tag{7.60}$$

If we now plug in our result from Eq. (7.56), recast as $h_N = 2h_{N'}$, this gives us:

$$I_{N'} - I_N = 15ch_{N'}^4 \tag{7.61}$$

But we already know that $\mathcal{E}_{N'} = ch_{N'}^4$ for Simpson's rule. Thus, we have been able to derive an expression for $\mathcal{E}_{N'}$ in terms of the two results I_N and $I_{N'}$:

$$\mathcal{E}_{N'} = \frac{1}{15}(I_{N'} - I_N) \tag{7.62}$$

This is good news: it means that we've been able to provide an estimate of the accuracy of our result $I_{N'}$ by using $I_{N'}$ itself (a calculation for N' panels) and I_N (a calculation for N panels). We've managed to provide a quantitative estimate of our ignorance, without knowing the correct answer ahead of time.

Something analogous ends up holding for the trapezoid rule:

$$\mathcal{E}_{N'} = \frac{1}{3}(I_{N'} - I_N) \tag{7.63}$$

and for the rectangle rule:

$$\mathcal{E}_{N'} = I_{N'} - I_N \tag{7.64}$$

as you will explicitly show. The only difference is in the denominator. Incidentally, a larger number in the denominator means that our absolute error is smaller: this is as it should be, since Simpson's rule is the most accurate method of the three.

7.3.2 Thoughts before Implementing

First, what do we do if going from N to $N' = 2N$ panels doesn't give us an absolute error that is small enough for our purposes? The answer is simple: we double again, and keep doing so until we have reached the accuracy we want. Notation-wise, you can envision schemes where you label the different iterations as N_i or something like that, but from a

programming perspective it's probably just easier to use N and $N' = 2N$ and then keep renaming your variables.

Second, how do we pick the starting value of N? This isn't too big a deal, either: simply start with any value you want (if you're using Simpson's rule, make sure that's an even number) and double it to $N' = 2N$ as described above. If you start with N too small, you might take too long (for your purposes) to meet your error tolerance. If you start with an N that is too large, you might be wasting your time if $\mathcal{E}_{N'}$ is already much smaller than you need.

Third, you should keep in mind that our integration routines were written in terms of n (the number of points), not N (the number of panels). Thus, if you want to calculate I_N and $I_{N'}$ using those routines you should make sure to pass in n and n' instead of N and N'. This is quite straightforward to do, too: for our starting value, we simply have $N = n - 1$, so we pass in $n = N + 1$. When we double the number of panels, $N' = 2N$, we don't quite double the number of points.[14] We simply carry out this elementary bookkeeping exercise: $N' = 2N$ combined with $N = n - 1$ and $N' = n' - 1$ leads to:

$$n' = 2n - 1 \tag{7.65}$$

As a trivial check, note that if n is odd, this expression gives an n' that is also odd.

7.3.3 Implementation

Code 7.2 is a Python implementation of an adaptive integrator. There are quite a few things to unpack here. First of all, note that the main program produces `ans`, the analytically known answer, but *only* so that we may print it out and compare at the end, i.e., this is not used anywhere inside the function we are defining.

Observe that we are employing a dictionary, using function names as the *keys*. Since functions are first-class objects in Python, we can use them as we would any other entity. This allows us to select a denominator value according to which integrating function is passed in as an argument *when we call* `adaptive()`! This is an incredibly powerful idiom: notice how the rest of the code simply uses `integrator()`, i.e., the code looks the same whether we want to create an adaptive integrator for the rectangle, trapezoid, or Simpson's rules (except for the value of the denominator). To appreciate how nice this is, you should try to implement your own version without looking at this solution: if you're still a beginner, you will likely end up having a cascade of `if` and `elif`, the bodies of which are carrying out the same operations by a different name. This is an invitation for trouble if, e.g., you later want to modify the algorithm, include an extra integrator, etc.

This code is making use of the `for-else` idiom, as usual; we set the return value to `None` if we are unsuccessful in determining the answer. Speaking of the error tolerance, our program sets that (arbitrarily) to 10^{-12}. We use a relative error ($\epsilon_{N'}$), not an absolute one ($\mathcal{E}_{N'}$). This is in order to make sure that we always conserve roughly 11 or 12 significant figures, but don't try to accomplish the impossible.

As you will see when you run this code, the rectangle rule fails to reach an absolute

[14] This would be an especially egregious mistake for Simpson's rule, since we need to start with an odd number of points n and also *end up* with an odd number of points n'.

Code 7.2 adaptive.py

```
from newtoncotes import f, rectangle, trapezoid, simpson
import numpy as np

def adaptive(f,a,b,integrator,kmax = 20,tol = 1.e-12):
     functodenom = {rectangle:1, trapezoid:3, simpson:15}
     denom = functodenom[integrator]

     n = 2
     val = integrator(f,a,b,n)
     for k in range(kmax):
          nprime = 2*n-1
          valprime = integrator(f,a,b,nprime)
          err = abs(valprime-val)/denom
          err /= abs(valprime)
          print(nprime, valprime, err)
          if err<tol:
               break

          n, val = nprime, valprime
     else:
          valprime = None
     return valprime

if __name__ == '__main__':
     ans = np.log(1 + np.sqrt(2))
     print(ans); print("")
     for integrator in (rectangle, trapezoid, simpson):
          print(adaptive(f, 0., 1., integrator)); print("")
```

error below our tolerance (thus returning `None`), even after doubling the number of panels 20 times. This is consistent with what we saw in Fig. 7.4: for a million points, the rectangle rule has an absolute error of order $\approx 10^{-7}$. The output for the trapezoid rule is analogous: we reach our tolerance levels with a bit over a quarter of a million points, just like in Fig. 7.4. Things are also fully analogous for Simpson's rule: 257 points (i.e., 256 panels) are enough to get the tiny absolute error we desire, highlighting the superiority of Simpson's rule.[15]

This output also serves to provide justification for our earlier claims, around Fig. 7.4,

[15] You shouldn't put too much faith in the Simpson's rule error bar for $n = 3$, as it follows from the Simpson's $n = 2$ answer and we already know that we should never apply Simpson's rule for an even number of points.

about the order-of-magnitude behavior of the error as h is decreased. For example, Simpson's rule (which never gets up to 1000 points) jumps by more than four orders magnitude when the number of panels goes from roughly 10 to roughly 100.

Before concluding, we note that here we have only presented a simple choice for an adaptive integration scheme: keep doubling the number of panels, until the answer for the integral has an acceptable error. In the implementation above, we throw away our earlier calculations, but in the problem set you will see how to be less wasteful. In any case, we have to keep re-doing the integral from a to b again and again. Another scheme might be to divide the interval a to b in two: we then have two separate integrals, $\int_a^{(a+b)/2} f(x)dx$ and $\int_{(a+b)/2}^b f(x)dx$, to carry out; however, we can take the tolerance in each subinterval to be one-half of the total desired tolerance. If that's not good enough, we further sub-divide, and continue doing this until we reach our desired sub-tolerance in each subinterval. This would be an especially good idea for a function that changes a lot in one region and is "boring" in another region; there's no point in doubling the number of your panels everywhere if you've already done a good enough job in a region where it's easy to integrate the function.

7.4 Romberg Integration

We now turn to the topic of Romberg integration, which results from the combination of the composite trapezoid rule together with Richardson extrapolation. The latter was first introduced in section 3.3.7, where we proceeded to apply it to the problem of estimating derivatives via finite differences. Here we do much the same, this time applied to numerical integration. Some of the equations that follow will be very similar to what we saw in section 7.3.1, when we were doubling the number of panels in the context of adaptive integration.

7.4.1 Richardson Extrapolation

We start from summarizing our earlier results from section 3.3.7 on Richardson extrapolation. The task there was to evaluate the quantity G. We made the assumption that the error term could be written as a sum of powers of h:

$$G = g(h) + Ah^p + Bh^{p+q} + Ch^{p+2q} + \cdots \tag{7.66}$$

where A, B, C are constants, p denotes the order of the leading error term, and q is the increment in the order for the error terms after that. The main idea behind Richardson extrapolation was to apply Eq. (7.66) twice, once for a step size h and once for a step size $h/2$. This gave us:

$$G = \frac{2^p g\left(\frac{h}{2}\right) - g(h)}{2^p - 1} + O(h^{p+q}) \tag{7.67}$$

Thus, starting from a formula with a leading error of $O(h^p)$, we used two calculations (one for a step size h and once for a step size $h/2$) and managed to eliminate the error $O(h^p)$. This

meant that we were left with a formula that has error $O(h^{p+q})$. At the time, we observed that this process could be repeated: by starting with two calculations each of which has error $O(h^{p+q})$, one can arrive at an answer with error $O(h^{p+2q})$, and so on. We will soon carry out precisely such a repetition.

Before we do that, we first re-express the main result of Richardson extrapolation in the language of integrals, using our new notation. Our task is to evaluate $I = \int_a^b f(x)dx$. The result for the value of the integral is I_N when using N panels. Remember that the panel width for this case is simply:

$$h_N = \frac{b-a}{N} \tag{7.68}$$

As above, we are interested in what happens when we start from a given panel number N and then change the panel number to $N' = 2N$. We saw in Eq. (7.56) that the new panel width $h_{N'}$ can be trivially related to the old panel width h_N, via $h_{N'} = h_N/2$. Thus, the Richardson-extrapolated estimate, as per Eq. (7.67), can be rewritten as:

$$I = \frac{2^p I_{N'} - I_N}{2^p - 1} + O(h_N^{p+q}) \tag{7.69}$$

where, as always, $N' = 2N$. This applies to a method for which the leading error goes as h_N^p: the extrapolation cancels that leading term, leaving us with an error of h_N^{p+q}.

In the spirit of being fully explicit, let us apply Eq. (7.69) for a few specific cases, which will come in handy in the next subsection. First, we examine the case of the trapezoid rule, where the absolute error behaves as:

$$\mathcal{E} = c_2 h_N^2 + c_4 h_N^4 + c_6 h_N^6 + \cdots \tag{7.70}$$

as per Eq. (7.41). In this case, Eq. (7.69) gives us:

$$I = \frac{2^2 I_{N'} - I_N}{2^2 - 1} + O(h_N^4) = \frac{4}{3} I_{N'} - \frac{1}{3} I_N + O(h_N^4) \tag{7.71}$$

where $N' = 2N$. Since the leading error goes as h_N^p, we have $p = 2$. Using two estimates, each of which has an error of order $O(h_N^2)$, we end up with an estimate with error $O(h_N^4)$.

Second, imagine we were dealing with a situation where the absolute error behaves as:

$$\mathcal{E} = c_4 h_N^4 + c_6 h_N^6 + c_8 h_N^8 + \cdots \tag{7.72}$$

In this case, Eq. (7.69) gives us:

$$I = \frac{2^4 I_{N'} - I_N}{2^4 - 1} + O(h_N^6) = \frac{16}{15} I_{N'} - \frac{1}{15} I_N + O(h_N^6) \tag{7.73}$$

where $N' = 2N$. Since the leading error goes as h_N^p, we have $p = 4$; employing two estimates with error of order $O(h_N^4)$, we get an estimate with error $O(h_N^6)$.

Third, imagine we were dealing with a situation where the absolute error behaves as:

$$\mathcal{E} = c_6 h_N^6 + c_8 h_N^8 + c_{10} h_N^{10} + \cdots \tag{7.74}$$

In this case, Eq. (7.69) gives us:

$$I = \frac{2^6 I_{N'} - I_N}{2^6 - 1} + O(h_N^8) = \frac{64}{63} I_{N'} - \frac{1}{63} I_N + O(h_N^8) \tag{7.75}$$

where $N' = 2N$. Since, the leading error goes as h_N^p, here we have $p = 6$. Thus, using two estimates with error $O(h_N^6)$, we end up with an estimate with error $O(h_N^8)$.

7.4.2 Romberg Recipe

Romberg integration amounts to using the trapezoid rule along with Richardson extrapolation. As discussed in the previous subsection, Richardson extrapolation involves doubling the number of panels, just like we did when discussing adaptive integration. In short, we will combine (a small number of) actual trapezoid-rule integrations with repeated Richardson-extrapolation steps. We note at this early stage that Romberg integration will give disappointing results when our function (or its derivatives) exhibits pathological behavior. Instead, we will only study here our standard example, which does not suffer from such problems.

Romberg integration uses some new notation, which can be confusing if you're not paying close attention. The confusion may arise from the fact that the Romberg approach involves some trapezoid rule results and some extrapolated results. In order to keep these distinct, we will be very explicit in our derivation in what follows. Start from a trapezoid-rule calculation that uses one panel. This was denoted by I_1 in the previous section. Introduce the new notation $R_{0,0} = I_1$: as a mnemonic, think of this as your starting point in two axes (which will be explained soon) the numbering for which starts at 0, as usual. This $R_{0,0}$ is the result of a real calculation, not an extrapolation (it's simply the trapezoid-rule result for one panel I_1). In order to carry out a Richardson extrapolation, we will need another "raw" calculation. Take the case of the trapezoid-rule result for two panels, denoted by I_2. Introduce the new notation $R_{1,0} = I_2$. Importantly, our new $R_{i,j}$ entity has two indices, and we've only increased the value of the first one. This will continue to be the case below every time we carry out a new trapezoid-rule computation: *whenever we carry out a new trapezoid rule step the first index in $R_{i,0}$ will go up by one and the second index will be 0.*

Now that we have access to the two trapezoid-rule results $R_{0,0}$ and $R_{1,0}$ (which we used to call I_1 and I_2) we can carry out a Richardson-extrapolation step. The situation we're faced with is identical to what we were facing in the "first case" in the previous subsection. We have at our disposal two results with errors that go as $c_2 h^2 + c_4 h^4 + c_6 h^6 + \cdots$ so we can apply Richardson extrapolation in the form of Eq. (7.71). Since we're already introducing new notation, let's keep at it. We call the result of this first extrapolation step $R_{1,1}$:

$$R_{1,1} = \frac{2^2 R_{1,0} - R_{0,0}}{2^2 - 1} = \frac{4}{3} R_{1,0} - \frac{1}{3} R_{0,0} \tag{7.76}$$

In this definition, we don't include the error term. Note how the second index on the left-hand side increased by one. This will continue to happen below: *whenever we carry out a Richardson-extrapolation step the second index in $R_{i,j}$ will go up by one.*

Our situation can be conveniently recast using matrices. Our two trapezoid-rule results

belong in the same column (the 0th one, since they both have the second index set to 0):

$$\begin{pmatrix} R_{0,0} \\ R_{1,0} \end{pmatrix} \tag{7.77}$$

As we saw in Eq. (7.76), carrying out a Richardson extrapolation allows us to move into a new column:

$$\begin{pmatrix} R_{0,0} & \\ R_{1,0} & R_{1,1} \end{pmatrix} \tag{7.78}$$

Importantly, both elements in the 0th column had a leading error term of the form $O(h^2)$, so they allowed us to produce a new estimate with a leading error term of the form $O(h^4)$, placed in the 1st column.

At this point, we decide to produce yet another "raw" calculation, namely the trapezoid-rule result for four panels, denoted by I_4 in the previous section. Since this is a new trapezoid result, we increase the first index in $R_{i,0}$ and store $R_{2,0} = I_4$, obtaining:

$$\begin{pmatrix} R_{0,0} & \\ R_{1,0} & R_{1,1} \\ R_{2,0} & \end{pmatrix} \tag{7.79}$$

It is still the case that all the results in the 0th column have an error of $O(h^2)$. However, that means that we are now free to perform yet another Richardson extrapolation: instead of combining $R_{0,0}$ and $R_{1,0}$ (which gave us $R_{1,1}$ above), we combine $R_{1,0}$ and $R_{2,0}$, to produce an answer which we store in $R_{2,1}$:

$$R_{2,1} = \frac{2^2 R_{2,0} - R_{1,0}}{2^2 - 1} = \frac{4}{3} R_{2,0} - \frac{1}{3} R_{1,0} \tag{7.80}$$

As our notation clearly shows, this new value belongs in the 1st column. It goes under $R_{1,1}$ and has an error $O(h^4)$ just like that value. But we now realize that we have access to two values in the 1st column, $R_{1,1}$ and $R_{2,1}$, each of which has a leading error of $O(h^4)$. The situation we're faced with is identical to the "second case" in the previous subsection. We have at our disposal two results with errors that go as $c_4 h^4 + c_6 h^6 + \cdots$ so we can apply Richardson extrapolation in the form of Eq. (7.73):

$$R_{2,2} = \frac{2^4 R_{2,1} - R_{1,1}}{2^4 - 1} = \frac{16}{15} R_{2,1} - \frac{1}{15} R_{1,1} \tag{7.81}$$

where you should observe that by combining two estimates with error $O(h^4)$ we have produced a new estimate with error $O(h^6)$. Thus, we have:

$$\begin{pmatrix} R_{0,0} & & \\ R_{1,0} & R_{1,1} & \\ R_{2,0} & R_{2,1} & R_{2,2} \end{pmatrix} \tag{7.82}$$

in matrix form. For reasons that will soon become clear, we now observe that we went from the matrix in Eq. (7.78) to that in Eq. (7.82) by carrying out three steps: (a) a new trapezoid

rule calculation, giving us $R_{2,0}$, (b) a Richardson extrapolation as per Eq. (7.80) giving us $R_{2,1}$, and (c) a Richardson extrapolation as per Eq. (7.81) giving us $R_{2,2}$. The most accurate answer at our disposal is currently in $R_{2,2}$, which is the bottom element on the diagonal.

You are probably starting to see how this works, but let's evaluate another row explicitly, to make things crystal clear. The 0th element in each new row is simply another "raw" calculation. In this case we will have the trapezoid-rule result for eight panels, denoted by I_8 in the previous section. Since this is a new trapezoid result, we increase the first index in $R_{i,0}$ and store $R_{3,0} = I_8$, obtaining:

$$\begin{pmatrix} R_{0,0} & & & \\ R_{1,0} & R_{1,1} & & \\ R_{2,0} & R_{2,1} & R_{2,2} & \\ R_{3,0} & & & \end{pmatrix} \tag{7.83}$$

As always, the results in the 0th column have an error of $O(h^2)$. Just like we did before, we will now employ the previous row and the 0th element of the current row to carry out several Richardson-extrapolation steps. First, we combine $R_{2,0}$ and $R_{3,0}$, to produce an answer which we store in $R_{3,1}$:

$$R_{3,1} = \frac{2^2 R_{3,0} - R_{2,0}}{2^2 - 1} = \frac{4}{3} R_{3,0} - \frac{1}{3} R_{2,0} \tag{7.84}$$

This new value belongs in the 1st column because it has an error $O(h^4)$. Second, we now realize that we have access to two values in the 1st column, $R_{2,1}$ and $R_{3,1}$, each of which has a leading error of $O(h^4)$. This means that we can carry out yet another Richardson-extrapolation step:

$$R_{3,2} = \frac{2^4 R_{3,1} - R_{2,1}}{2^4 - 1} = \frac{16}{15} R_{3,1} - \frac{1}{15} R_{2,1} \tag{7.85}$$

Thus, by combining two estimates with error $O(h^4)$ we have produced a new estimate with error $O(h^6)$, which belongs in the 2nd column. Third, we are now able to carry out yet another step because the situation we're faced with is identical to the "third case" in the previous subsection. We have at our disposal two results with errors that go as $c_6 h^6 + c_8 h^8 + \cdots$ so we can apply Richardson extrapolation in the form of Eq. (7.75):

$$R_{3,3} = \frac{2^6 R_{3,2} - R_{2,2}}{2^6 - 1} = \frac{64}{63} R_{3,2} - \frac{1}{63} R_{2,2} \tag{7.86}$$

This new estimate belongs in the 3rd column because it has an error $O(h^8)$. Thus, we have:

$$\begin{pmatrix} R_{0,0} & & & \\ R_{1,0} & R_{1,1} & & \\ R_{2,0} & R_{2,1} & R_{2,2} & \\ R_{3,0} & R_{3,1} & R_{3,2} & R_{3,3} \end{pmatrix} \tag{7.87}$$

in matrix form. We now reiterate our main point about how to produce a new row in this matrix, by observing how we went from the matrix in Eq. (7.82) to that in Eq. (7.87).

First, we doubled the number of panels and carried out a new trapezoid-rule calculation and, then, we carried out three Richardson-extrapolation steps in Eq. (7.84), Eq. (7.85), and Eq. (7.86). The most accurate answer at our disposal is currently in $R_{3,3}$, which is the bottom element on the diagonal.

In summary, to produce a new row we first carry out a new trapezoid-rule calculation for twice as many panels as before (placing that in the 0th column) and then carry out Richardon extrapolation steps *which require the previous row and the newly produced 0th-column trapezoid result*. This leads to a lower-triangular matrix. The 0th column contains trapezoid-rule results, which therefore have error $O(h^2)$. The 1st column contains results for one Richardson-extrapolation step, which therefore have error $O(h^4)$. The 2nd column contains results of two Richardson-extrapolation steps, which therefore have error $O(h^6)$. The pattern should be clear by now. As you go down the 0th column the number of panels is doubled so the h becomes smaller. As you go to the right on any row you carry out increasingly more Richardson-extrapolation steps. As a result, the bottom element on the diagonal is the best estimate each time, so we compare our best value on this row with our best value on the previous row and terminate this process when the difference between these two "best estimates" is smaller than some set tolerance level. In equation form:

$$\begin{aligned} R_{i,0} &= I_N && (N = 2^i, i = 0, 1, 2, \ldots) \\ R_{i,j} &= \frac{4^j R_{i,j-1} - R_{i-1,j-1}}{4^j - 1} && (i = 1, 2, \ldots \text{ and } j = 1, 2, \ldots, i) \end{aligned} \tag{7.88}$$

Let's try to unpack this. The first line simply says that the elements of the 0th column are trapezoid-rule results, with the number of panels being doubled as we move down the column: $R_{0,0}$ is I_1, $R_{1,0}$ is I_2, $R_{2,0}$ is I_4, $R_{3,0}$ is I_8, and so on.[16] The second line simply encapsulates how we carry out the Richardson extrapolation: we always use (a) the element that's in the previous column on the same row, and (b) the element that's in the previous column on the previous row. This equation on the second line gives us all the elements of the lower-triangular matrix except for the 0th column. It's also worth observing that the second line in this pair of equations uses 4^j for the coefficient in the numerator and the denominator. This is simply a result of noticing that the powers that show up are 2^2, 2^4, 2^6, and so on, which are conveniently re-expressed by remembering that $(2^2)^j = 4^j$.

As you can imagine from the expected scaling with h as you move to the right of the matrix, Romberg integration is a nice way of starting from a crude integration method (the trapezoid rule) and producing very accurate results with very few function evaluations (as usual, assuming the integrand is not pathological). As a result, the very simple trapezoid rule ends up being the workhorse of some very accurate calculations.

Before we turn to a Python implementation, applied to a specific problem, we note a general feature of Romberg integration. This is that the 1st column (i.e., the one containing results of one Richardson-extrapolation step) of the Romberg matrix turns out to be

[16] Note that we've avoided the use of the confusing-looking I_{2^i}.

identical to Simpson's rule results. To be more specific, as you will show in a problem, $R_{i,1}$ contains the Simpson's 1/3 answer for 2^i panels, for $i = 1, 2, \ldots$.

7.4.3 Implementation

Thinking about the simplest way of implementing Romberg integration, a few things immediately come to mind. First, since the 0th column elements will be trapezoid-rule results while doubling the number of panels, it seems reasonable to use the adaptive-integration code `adaptive.py` as a starting point. Second, in addition to carrying out these trapezoid-rule calculations, we'll also need to carry out the Richardson-extrapolation steps shown in the second line of Eq. (7.88). Since these implement a specific equation, it stands to reason that we should define a separate function encapsulating that process. What should that use as parameters and as a return value? As already emphasized, to carry out the sequence of Richardson-extrapolation steps that fill out an entire row, what we need (as parameters) are the previous row and the newly produced 0th-column trapezoid estimate. The question naturally arises as to how to store the $R_{i,j}$. Since these make up a matrix, it seems natural to reach for a `numpy` array. However, we don't really need to store the entire matrix: at any point in time, all we need is the current row and the previous one, so we can simply use two (one-dimensional) lists.

The above considerations lead to the implementation in Code 7.3. We skip (for now) the `prettyprint()` function. As advertised, this code includes a function `richardson()` which implements the second line of Eq. (7.88). It takes in a list containing the previous row (`Rprev`) and a float containing the 0th element on the current row (`Rincurr0`).

As promised, the major function here, `romberg()`, merely keeps track of the current row and the previous one (`Rcurr` and `Rprev`, respectively), not the entire matrix. The doubling of the panels (implemented via `n` and `nprime`) is identical to what we had in the earlier code. This code calls `trapezoid()` whenever a new 0th-column element needs to be produced and `richardson()` for everything else. Note that the loop starting value is different than in the older code, since `i` actually matters here: it is passed in as an argument to `richardson()`.

The error is estimated by using the last element on the current and the previous row. Note that there's no denominator here, since we are not employing an analytical expectation of how these two elements are related: we simply expect that when the last diagonal elements stop changing our work is done.

The only other point is the line saying `Rprev = Rcurr[:]` whose logic should be familiar enough by now: in the next iteration our current row will be treated as the previous row. Copying the entire list (as opposed to renaming it) is not really necessary here (since a new list is created inside `richardson()` each time), but it's a good habit to maintain.

Running this code, we notice that the (absolute) error of our estimate is 10^{-11}, even though we only hard-coded a requirement for a (relative) error of 10^{-8} inside `romberg()`. This is good but, without knowing how many function evaluations were necessary, not necessarily impressive. To find out how good our approach is, we could print out the value of `nprime` as we did before. Instead, we could just uncomment the two lines containing

Code 7.3 romberg.py

```
from newtoncotes import f, trapezoid
import numpy as np

def prettyprint(row):
     for elem in row:
          print("{0:1.10f} ".format(elem),end="")
     print("")

def richardson(Rprev, Rincurr0, i):
     Rcurr = [Rincurr0]
     for j in range(1, i+1):
          val = (4**j*Rcurr[j-1] - Rprev[j-1])/(4**j - 1)
          Rcurr.append(val)
     return Rcurr

def romberg(f,a,b,imax = 20,tol = 1.e-8):
     n = 2
     val = trapezoid(f,a,b,n)
     Rprev = [val]
     #prettyprint(Rprev)
     for i in range(1,imax):
          nprime = 2*n-1
          Rincurr0 = trapezoid(f,a,b,nprime)
          Rcurr = richardson(Rprev, Rincurr0, i)
          #prettyprint(Rcurr)
          err = abs(Rprev[-1] - Rcurr[-1])/abs(Rcurr[-1])
          valprime = Rcurr[-1]
          if err < tol:
               break
          n = nprime
          Rprev = Rcurr[:]
     else:
          valprime = None
     return valprime

if __name__ == '__main__':
     ans = np.log(1 + np.sqrt(2))
     print(ans)
     print(romberg(f,0.,1.))
```

calls to our beautifying `prettyprint()` function in the present code to get all the $R_{i,j}$'s that came into the picture:

```
0.8535533906
0.8739902908 0.8808025909
0.8795307704 0.8813775970 0.8814159307
0.8809131418 0.8813739323 0.8813736880 0.8813730175
0.8812584924 0.8813736093 0.8813735877 0.8813735861 0.8813735884
0.8813448144 0.8813735884 0.8813735870 0.8813735870 0.8813735870 0.8813735870
```

Here, the first row corresponds to a trapezoid-rule calculation with one panel, I_1. Going down the 0th column we then encounter I_2, I_4, I_8, I_{16}, and finally I_{32}. This means that in order to arrive at our very accurate final result (the rightmost one in the last row) we only needed to carry out 33 function evaluations (32 panels means 33 points). This is truly spectacular, if you compare to the output of `adaptive.py`, where an error of 10^{-11} for the trapezoid rule needed 65 537 points!

Well, not quite. As it so happens, our implementation of the Romberg recipe is wasteful: it throws away the older trapezoid calculations when producing the new ones (e.g., the function evaluations used to produce I_8 are discarded when evaluating I_{16}). As a result, we actually carried out $2 + 3 + 5 + 9 + 17 + 33 = 69$ function evaluations. Still, that's roughly a factor of a thousand fewer function evaluations than the non-Romberg trapezoid approach. As a matter of fact, even Simpson's rule needed roughly 100 points to produce an estimate with an absolute error of 10^{-11}. In a problem, you will write and test your own non-wasteful Romberg integration code.

7.5 Gaussian Quadrature

The Newton–Cotes formulas we encountered have been cast in the same form, that of Eq. (7.7), where we were dealing with n nodal abscissas x_i and n weights c_i. All of these Newton–Cotes methods employ an evenly spaced grid, as per Eq. (7.8). In other words, such methods all use pre-determined nodal abscissas x_i and differ only in which weights c_i they employ; the latter are chosen in order to exactly integrate a low-degree polynomial. In contradistinction to this, what the method(s) of Gaussian quadrature accomplish is both simple and impressive: instead of limiting yourself to equally spaced nodal abscissas, *expand your freedom so that both the n abscissas x_i and the n weights c_i are at your disposal*. Then, you will be able to *integrate all polynomials up to degree* $2n - 1$ *exactly*! It should be straightforward to see why this is impressive: it implies that even using a very small number of points (say $n = 5$) allows us to integrate all polynomials up to quite high order exactly (up to ninth order for $n = 5$). As you may have already guessed, the reason this is possible is that we have doubled the number of parameters at our disposal: we can use the $2n$ parameters (the x_i's and the c_i's) to handle polynomials up to degree $2n - 1$.

In our discussion of Newton–Cotes methods, we always started from an elementary

interval and later generalized to a composite integration rule, which could handle the full integral from a to b. Given the accuracy that Gaussian quadrature is capable of, we will employ it *directly* in the full interval from a to b.[17] Just like for the problem of interpolation (chapter 6), using unequally spaced abscissas allows you to eliminate the issues that arise with an equally spaced grid; as a result, you can use a single polynomial interpolant for the entire interval instead of employing piecewise polynomial interpolation as we did for Newton–Cotes integration above.

It is standard to take $a = -1$ and $b = 1$ at this stage: this is completely analogous to what we did in our discussion of polynomial interpolation in section 6.2. Thus, we start by focusing on the standard interval $[-1, 1]$; as we saw in Eq. (6.13) and will discuss again below, it is straightforward to scale to the interval $[a, b]$ after the fact. It is important to understand that our use of the standard interval $[-1, 1]$ is unrelated to an elementary interval: as already mentioned, we will set up Gauss–Legendre directly on the full interval (which happens to be $[-1, 1]$), i.e., we will not first study an elementary rule and later a composite rule. Thus, our defining equation will be:

$$\int_{-1}^{1} f(x)dx \approx \sum_{k=0}^{n-1} c_k f(x_k) \tag{7.89}$$

Our earlier claim was that by an intelligent choice of *both* the x_k's and the c_k's, we will be able to integrate polynomials up to degree $2n - 1$ exactly. It's probably not too early to point out that *all Gaussian quadrature methods are open*, so the x_k's are *not* to be identified with -1 and $+1$. Let's see how this all works.

7.5.1 Gauss–Legendre: $n = 2$ Case

We start from explicitly addressing the exact integrability of polynomials up to degree $2n - 1$ for the simplest non-trivial case, namely that of two points.[18] We take Eq. (7.89) and apply it to the case of $n = 2$:

$$\int_{-1}^{1} f(x)dx = c_0 f(x_0) + c_1 f(x_1) \tag{7.90}$$

The abscissas x_0 and x_1 along with the weights c_0 and c_1 are the four quantities we need to now determine. Since Gaussian quadrature methods are open, x_0 and x_1 will lie in (a, b). As advertised, we will determine the four unknown parameters by demanding that all polynomials up to order $2n - 1 = 3$ can be integrated exactly.

It is simplest to use monomials, i.e., single powers of x, instead of polynomials (which

[17] In practice, one sometimes slices up the integration interval "by hand", using more points in regions that exhibit more structure, and fewer points when the behavior of the integrand is pretty simple.

[18] A problem invites you to study the even more basic case of $n = 1$, which is equivalent to the midpoint rule!

can later be arrived at as linear combinations of monomials). Thus, we will take:

$$f(x) = \begin{cases} x^0 \\ x^1 \\ x^2 \\ x^3 \end{cases} \tag{7.91}$$

and assume that for all these cases Eq. (7.90) holds. This leads to the following four equations in four unknowns:

$$\begin{aligned} &\int_{-1}^{1} 1dx = 2 = c_0 + c_1, && \int_{-1}^{1} xdx = 0 = c_0 x_0 + c_1 x_1, \\ &\int_{-1}^{1} x^2 dx = \frac{2}{3} = c_0 x_0^2 + c_1 x_1^2, && \int_{-1}^{1} x^3 dx = 0 = c_0 x_0^3 + c_1 x_1^3 \end{aligned} \tag{7.92}$$

In each of these cases the left-hand side is that of Eq. (7.90), the second step gives the answer for that integral (calculated analytically), and the third step gives the right-hand side of Eq. (7.90), applied to that specific monomial.

We are now faced with a set of nonlinear equations. The second of these gives us $c_0 x_0 = -c_1 x_1$; this result, when combined with the fourth relation, leads to $x_0^2 = x_1^2$. Thus, since x_0 and x_1 are different from each other, we find $x_0 = -x_1$. Together with the second relation, this implies $c_0 = c_1$. This result, together with the first relation in Eq. (7.92) gives $c_0 = c_1 = 1$. The third relation in Eq. (7.92) now gives us $x_0^2 + x_0^2 = 2/3$, which can be solved by taking $x_0 = -1/\sqrt{3}$. Thus, we have arrived at the solution:

$$c_0 = c_1 = 1, \qquad x_0 = -\frac{1}{\sqrt{3}}, \qquad x_1 = \frac{1}{\sqrt{3}} \tag{7.93}$$

This can now be used to recast Eq. (7.90) as:

$$\int_{-1}^{1} f(x)dx \approx f\left(-\frac{1}{\sqrt{3}}\right) + f\left(\frac{1}{\sqrt{3}}\right) \tag{7.94}$$

This employs two points and is exact for polynomials up to third order; it is approximate for other functions. It's worth pausing for a second to realize that our accomplishment is already noteworthy: Simpson's rule could integrate polynomials up to third order exactly, but it needed at least three points in order to do that.

7.5.2 Gauss–Legendre: General Case

There are several ways one could go about introducing Gaussian quadrature. First, you could try to generalize the approach of the previous section to the case of larger n. As you can imagine, this will lead to a system of increasingly nonlinear equations; this is precisely the task we tackled in one of the problems of chapter 5. This approach to Gauss–Legendre integration involves a Vandermonde matrix and is therefore increasingly unstable as n gets larger. Second, one could employ the standard-textbook approach, which

introduces the theory of orthogonal polynomials by fiat right about now, typically leaving students scratching their heads. This involves a number of auxiliary entities and is almost never carried through to the end, resorting instead to the dreaded phrase "it can be shown"; for the sake of completeness, we briefly go over that alternative argument in one of the problems. Intriguingly, there exists a third approach to implementing Gauss–Legendre quadrature, known as the *Golub–Welsch algorithm*; this involves Jacobi matrices, which were touched upon in a problem in chapter 5 and are further elaborated on in the present chapter's problem set. This happens to be a robust approach which is employed in state-of-the-art libraries, but it assumes one is already convinced of the importance of orthogonal polynomials to this problem.

In contradistinction to these approaches, in the present section we will follow a fourth option, tackling Gaussian quadrature using *Hermite interpolation*, a subject we introduced in section 6.2.4. As you may recall, Hermite interpolation arises when you wish to interpolate through both function and derivative values. At the simplest level, Hermite interpolation is the only tool we've encountered that can handle polynomials of degree $2n-1$. While the question of exactly how Hermite interpolation accomplishes the task at hand may initially be a bit nebulous, everything will become clear soon enough and will then follow straightforwardly from this one choice. Crucially, our approach will allow us to explain in a unified manner: (a) how to pick the nodes x_k, thereby justifying the presence of the name "Legendre" in Gauss–Legendre, (b) how to compute the weights c_k, and (c) the overall error scaling of Gauss–Legendre quadrature. We discuss these topics in turn, in the following subsections.

Node Placement

In section 7.2.3 we saw how to produce Newton–Cotes quadrature rules starting from Lagrange interpolation, writing the interpolating polynomial $p(x)$ of degree $n-1$ in terms of the function values at the grid points, as well as the cardinal polynomials, see Eq. (7.24). We now write down the analogous formula for the case of Hermite interpolation, since we are interested in studying polynomials of degree up to $2n-1$. This is Eq. (6.40):

$$p(x) = \sum_{k=0}^{n-1} f(x_k)\alpha_k(x) + \sum_{k=0}^{n-1} f'(x_k)\beta_k(x) \tag{7.95}$$

where we have updated the notation to employ $f(x_k)$ and $f'(x_k)$ for our integrand and derivative values at the nodes; the $\alpha_k(x)$ and $\beta_k(x)$ are determined in terms of the cardinal polynomials as per Eq. (6.49). You may be worried about the fact that we are trying to interpolate through the $f'(x_k)$ values, which we don't actually know; our goal is to express an integral in terms of function values as per Eq. (7.89), not in terms of function-derivative values. For now, you should be patient.

We are actually interested in integrating $f(x)$: if we pick $f(x)$ to be a polynomial of degree $2n-1$ or less, then Eq. (7.95) will *exactly* match $f(x)$. To see this, remember that our error formula for Hermite interpolation from Eq. (6.50) involved a $2n$-th derivative, $f^{(2n)}$. For a polynomial of degree $2n-1$ or less, that derivative is always zero, implying

that $p(x)$ perfectly matches $f(x)$ for this case. We are therefore free to integrate the $p(x)$ of Eq. (7.95) from −1 to +1:

$$\begin{aligned}\int_{-1}^{1} f(x)dx = \int_{-1}^{1} p(x)dx &= \sum_{k=0}^{n-1} f(x_k)\int_{-1}^{1}\alpha_k(x)dx + \sum_{k=0}^{n-1} f'(x_k)\int_{-1}^{1}\beta_k(x)dx \\ &= \sum_{k=0}^{n-1} c_k f(x_k) + \sum_{k=0}^{n-1} d_k f'(x_k) \end{aligned} \tag{7.96}$$

In the last step we defined:

$$c_k = \int_{-1}^{1}\alpha_k(x)dx, \qquad d_k = \int_{-1}^{1}\beta_k(x)dx \tag{7.97}$$

If we can now somehow make the d_k's vanish, we will have accomplished what we set out to: the left-hand side of Eq. (7.96) is the integral we wish to evaluate and the last step in Eq. (7.96) would express it (*exactly*) as a sum of weights times function values, precisely as in Eq. (7.89). Also, if the d_k's dropped out of the problem then we would no longer have to worry about the fact that we don't know the $f'(x_k)$ values.

To see how we could possibly make the d_k's vanish, we express the $\beta_k(x)$ in terms of the cardinal polynomials as per Eq. (6.49):[19]

$$d_k = \int_{-1}^{1}\beta_k(x)dx = \int_{-1}^{1}(x-x_k)L_k^2(x)dx = \frac{1}{L'(x_k)}\int_{-1}^{1}L(x)L_k(x)dx \tag{7.98}$$

The x_k in the second step are the interpolation nodes and similarly $L_k(x)$ are the cardinal polynomials defined for those nodes. In the last step, we expressed (one of) the $L_k(x)$ in terms of the node polynomial and its derivative, as per Eq. (6.27); we pulled the $L'(x_k)$ outside the integral since it doesn't depend on x.

We now recall the definition of the node polynomial from Eq. (6.24):

$$L(x) = \prod_{j=0}^{n-1}(x-x_j) \tag{7.99}$$

This is a monic polynomial in x of degree n, that vanishes at the interpolation nodes x_j. Similarly, we remember the definition of a cardinal polynomial from Eq. (6.19):

$$L_k(x) = \frac{\prod_{j=0,j\neq k}^{n-1}(x-x_j)}{\prod_{j=0,j\neq k}^{n-1}(x_k-x_j)} \tag{7.100}$$

This is a polynomial in x of degree $n-1$. The way to make progress is to demand that the integral in Eq. (7.98) vanishes: we recall, from the problem on Gram–Schmidt orthogonalization for functions in chapter 4, that we had derived:

$$(r_{n-1}, q_n) \equiv \int_{-1}^{1} r_{n-1}q_n dx = 0 \tag{7.101}$$

[19] Since we're integrating over the entire interval, we're using entities like $L_k(x)$, rather than the more awkward notation of $L_{i+j}(x)$ in Eq. (7.25).

in Eq. (4.338). In this equation, $r_{n-1}(x)$ was a general polynomial of degree $n-1$ and q_n was an orthogonal polynomial of degree n. This would be precisely the situation we are faced with in Eq. (7.98) *if* $L(x)$ was an orthogonal polynomial. However, we recall that the interpolation nodes, i.e., the x_k's, are still at our disposal: *if we take them to be the zeros of the orthogonal polynomial of degree n*, then $L(x)$ will be the unique[20] monic polynomial of degree n that is orthogonal (in the interval $[-1,+1]$) to all polynomials of degree $n-1$ or less. In that case all the d_k's will vanish and Eq. (7.96) will take the form:

$$\int_{-1}^{1} f(x)dx = \sum_{k=0}^{n-1} c_k f(x_k) \tag{7.102}$$

where the c_k can be determined from Eq. (7.97). As the preceding derivation shows, this relation is *exact* when $f(x)$ is a polynomial of degree up to $2n-1$ and the x_k's are taken to be the roots of the orthogonal polynomial of degree n. As noted, there is no error term here, since the $2n$-th derivative, $f^{(2n)}$, vanishes. Similarly, there is no term involving $f'(x_k)$ since the d_k's vanish. In short, we have managed to employ Hermite interpolation to integrate polynomials of degree $2n-1$ or less using n nodal abscissas, with the derivatives dropping out of the problem.[21] Crucially, this was a result following from *orthogonality*, i.e., *integration*: the d_k's (which involve an integral) vanish, whereas the $\beta_k(x)$'s (which appear in Hermite interpolation) do not vanish (outside the interpolation nodes).

At this point, we remember that in the chapter 4 problem on Gram–Schmidt orthogonalization we had observed that the orthonormal q_j we had been finding there were multiples of the orthogonal *Legendre polynomials*, $P_n(x)$. The same holds for our monic orthogonal polynomials $L(x)$: with the exception of an overall multiplicative constant, $P_n(x)$ is the same polynomial as $L(x)$. We actually computed this multiplicative factor in Eq. (3.90), using Rodrigues' formula:

$$P_n(x) = \frac{(2n)!}{2^n (n!)^2} L(x) \tag{7.103}$$

Most importantly for our purposes, the *zeros of* $L(x)$ *are the zeros of the Legendre polynomial* $P_n(x)$ since, as we mentioned at the end of section 5.3, you will still find the same roots if you multiply a polynomial with a number. As it so happens, we've already written a Python code that computes the zeros of Legendre polynomials, `legroots.py`. You can now see why we have been speaking of Gauss–*Legendre* quadrature: this is Gaussian quadrature when the nodal abscissas are taken to be the roots of Legendre polynomials. This also explains why we decided to focus on the interval $[-1,+1]$ in the first place.

The main advantage of our derivation is that the theory of orthogonal polynomials (and the placement of the nodes at their roots) occurs organically: we didn't start the section using Legendre polynomials, but these emerged naturally when we needed to make the d_k's vanish, thereby managing to integrate exactly all polynomials of degree up to $2n-1$.

[20] A problem guides you toward showing this uniqueness property.
[21] Thus, the input is not x_k, $f(x_k)$, and $f'(x_k)$, but solely x_k and $f(x_k)$.

It's worth observing that in this one subsection we have employed Legendre polynomials from chapter 2, their leading coefficient from chapter 3, their orthogonality from chapter 4, their zeros from chapter 5, as well as Hermite interpolation and cardinal polynomials from chapter 6. This is a prominent example of the fact that the presentation in this book is cumulative: we didn't need to state anything without proof, but you will need to master the earlier material in order to grasp how everything fits together.

Weight Computation

After deriving Eq. (7.102), we observed that the c_k's can be determined from Eq. (7.97), which in its turn can make use of Eq. (6.48); this is true, and is an avenue you will explore in a problem; here, we realize that we can easily arrive at a simpler expression. Our Eq. (7.102) holds for any polynomial of degree up to $2n-1$, which means it will also hold for $f(x) = L_j(x)$: since $L_j(x)$ is a polynomial of degree $n-1$, it can be exactly integrated via Gauss–Legendre quadrature. Thus:

$$\int_{-1}^{1} L_j(x)dx = \sum_{k=0}^{n-1} c_k L_j(x_k) = \sum_{k=0}^{n-1} c_k \delta_{jk} \tag{7.104}$$

where the last step followed from our general result on cardinal polynomials, Eq. (6.21). Thus, we have been able to show that:

$$c_j = \int_{-1}^{1} L_j(x)dx \tag{7.105}$$

Intriguingly, this is formally identical to what we had found for Newton–Cotes integration, see Eq. (7.27). The only difference is that here we are not integrating over an elementary interval, but directly over the full interval.

For the Newton–Cotes case, we studied only small n values, given that we were only interested in an elementary interval, so we were able to explicitly carry out the integrals giving the weights for the trapezoid and Simpson's rules. In the Gauss–Legendre case, we're keeping n general, so we should try to see if we can re-express the integral involved. It is straightforward to start doing so:

$$c_j = \int_{-1}^{1} L_j(x)dx = \int_{-1}^{1} \frac{L(x)}{(x-x_j)L'(x_j)}dx = \frac{1}{P_n'(x_j)} \int_{-1}^{1} \frac{P_n(x)}{x-x_j}dx \tag{7.106}$$

In the second step we expressed the cardinal polynomial in terms of the node polynomial and its derivative, as per Eq. (6.27), as in Eq. (7.98). In the third step we remembered that $L(x)$ is simply a multiple of the Legendre polynomial $P_n(x)$, as per Eq. (7.103); since the multiplicative factor appears on both the numerator and the denominator, it cancels.

This is progress, but we're not quite there yet: Eq. (7.106) involves an integral, which we cannot do numerically, since that's why we're computing the c_j's in the first place. To

proceed, we invoke the *Christoffel–Darboux identity* for Legendre polynomials:

$$\sum_{k=0}^{n}(2k+1)P_k(x)P_k(y) = (n+1)\,\frac{P_{n+1}(x)P_n(y) - P_n(x)P_{n+1}(y)}{x-y} \tag{7.107}$$

which you are guided toward deriving in a problem. We now pick $y = x_j$, where x_j is a root of $P_n(x)$, i.e., $P_n(x_j) = 0$. That makes the n-th term in the sum on the left-hand side as well as the first term in the numerator on the right-hand side vanish, leaving us with:

$$\sum_{k=0}^{n-1}(2k+1)P_k(x_j)P_k(x) = -(n+1)P_{n+1}(x_j)\,\frac{P_n(x)}{x-x_j} \tag{7.108}$$

We now recall from Eq. (2.79) that $P_0(x) = 1$, so we can think of the left-hand side of Eq. (7.108) as already containing $P_0(x)$. We can then integrate from -1 to $+1$ to find:

$$\sum_{k=0}^{n-1}(2k+1)P_k(x_j)\int_{-1}^{1}P_0(x)P_k(x)dx = -(n+1)P_{n+1}(x_j)\int_{-1}^{1}\frac{P_n(x)}{x-x_j}dx \tag{7.109}$$

At this point, we employ the orthogonality of Legendre polynomials, which has the form:

$$(P_n(x), P_m(x)) \equiv \int_{-1}^{1}P_n(x)P_m(x)dx = \frac{2}{2n+1}\delta_{nm} \tag{7.110}$$

as we saw in Eq. (4.339). Using this on the left-hand side of Eq. (7.109) we get:

$$2 = -(n+1)P_{n+1}(x_j)\int_{-1}^{1}\frac{P_n(x)}{x-x_j}dx \tag{7.111}$$

where we dropped the $P_0(x_j)$ on the left-hand side, since it's equal to 1.

We now realize that the integral in Eq. (7.111) is the same as that appearing in Eq. (7.106); putting these two equations together leads to:

$$c_j = -\frac{2}{(n+1)P_{n+1}(x_j)P'_n(x_j)} \tag{7.112}$$

This is, in principle, the end of our journey: as you may recall, `legendre.py` computed not only the value of the Legendre polynomial, but also the value of its first derivative. Thus, we could now proceed to a Python implementation of Eq. (7.112). However, this formula involves a Legendre polynomial and its derivative for two different degrees; let us further simplify our expression for the weights. From Eq. (2.89) we get:

$$(1-x^2)P'_n(x) = -nxP_n(x) + nP_{n-1}(x) \tag{7.113}$$

We can also write down Eq. (2.86) for the case of $j = n$:

$$(n+1)P_{n+1}(x) = (2n+1)xP_n(x) - nP_{n-1}(x) \tag{7.114}$$

Adding the last two equations together, we eliminate $nP_{n-1}(x)$ to find:

$$(1-x^2)P'_n(x) = (n+1)xP_n(x) - (n+1)P_{n+1}(x) \tag{7.115}$$

If we now apply this relation at $x = x_j$, where as usual $P_n(x_j) = 0$, we get:

$$(1-x_j^2)P'_n(x_j) = -(n+1)P_{n+1}(x_j) \tag{7.116}$$

which, combined with Eq. (7.112), gives for the *weights in Gauss–Legendre quadrature*:

$$c_j = \frac{2}{(1 - x_j^2)[P_n'(x_j)]^2} \tag{7.117}$$

As desired, this requires the value of the Legendre polynomial (derivative) of a single degree. The plan of action, were one to implement things programmatically at this point, should be clear: Gauss–Legendre quadrature amounts to employing Eq. (7.102), where the x_k's are produced by finding the roots of $P_n(x)$ and the c_k's are computed using Eq. (7.117).

We now produce an identity which must be obeyed by the Gauss–Legendre weights. We have showed that Eq. (7.102) holds for all polynomials of degree $2n - 1$ or less, so it also holds for the case of a constant polynomial, $f(x) = 1$. This leads to:

$$\sum_{k=0}^{n-1} c_k = 2 \tag{7.118}$$

The 2 on the right-hand side is the result of working in the standard interval; it would have been $b - a$ in the general case. As mentioned after Eq. (7.29), in one of the problems you will show something analogous for Newton–Cotes methods. Speaking of the weights and Newton–Cotes, there is another difference between those methods and Gaussian quadrature ones: as you will show in another problem, in Gauss–Legendre integration *all the weights are positive*. This turns out to be different from the behavior of Newton–Cotes weights, where for higher degrees we tend to see both positive and negative weights (bringing to mind the finite-difference formulas from chapter 3 and the accompanying error issues).

Error Analysis

Similarly to what we did in the case of the Newton–Cotes rules, we will now discuss the error scaling in Gauss–Legendre quadrature. More specifically, just as we did for the trapezoid rule near Eq. (7.33) we will start from an interpolation error formula and then will integrate that. Of course, this time we are dealing with Hermite (not Lagrange) interpolation, and we are also dealing with the full (not elementary) interval.

In our derivation of Gauss–Legendre quadrature above, we were dealing with polynomials of degree up to $2n - 1$, which is why there was no error term in Eq. (7.96). We will now address the general case, where our interpolating polynomial $p(x)$ of degree up to $2n - 1$ may not fully capture the function $f(x)$, which could be non-polynomial (or of higher degree). Let us start from the Hermite-interpolation error formula, Eq. (6.50):

$$f(x) = p(x) + \frac{f^{(2n)}(\zeta)}{2n!} \prod_{j=0}^{n-1} (x - x_j)^2 \tag{7.119}$$

where we slightly updated the notation. We move the $p(x)$ to the left-hand side and identify

the product over $(x - x_j)^2$ with the square of the node polynomial from Eq. (6.24):

$$f(x) - p(x) = \frac{f^{(2n)}(\zeta)}{2n!} \prod_{j=0}^{n-1} L^2(x) \tag{7.120}$$

Integrating this difference from -1 to $+1$ gives us the error in Gauss–Legendre quadrature:

$$\mathcal{E} = \int_{-1}^{1} \frac{f^{(2n)}(\zeta)}{2n!} L^2(x)dx = \frac{f^{(2n)}(\xi)}{2n!} \int_{-1}^{1} L^2(x)dx \tag{7.121}$$

In the second step we used the extended version of the integral mean-value theorem to effectively pull out the derivative term. Similarly to the situation in Eq. (7.34), this is here justified because $L^2(x)$ is non-negative.

If we now express the node polynomial $L(x)$ in terms of the corresponding Legendre polynomial $P_n(x)$ as per Eq. (7.103), we will be left with an integral over $P_n^2(x)$. But that, in its turn, can be straightforwardly carried out using the orthogonalization/normalization relation for Legendre polynomials from Eq. (7.110). Putting it all together, we find the following error formula for Gauss–Legendre quadrature:

$$\mathcal{E} = \frac{2^{2n+1}(n!)^4}{(2n+1)[(2n)!]^3} f^{(2n)}(\xi) \tag{7.122}$$

This is more inscrutable than the corresponding formulas for Newton–Cotes methods, for two reasons: first, the $2n$-th derivative is typically hard to estimate and, second, all the powers and factorials make it hard to intuitively grasp how the error scales with n.

To elucidate things, we take one of the $(2n)!$ terms from the denominator and group it with the derivative term. The remaining prefactor can be approximated as follows:[22]

$$\frac{2^{2n+1}(n!)^4}{(2n+1)[(2n)!]^2} \sim \frac{\pi}{4^n} \tag{7.123}$$

Using this asymptotic relation, we can now give our final expression for the (approximation) *error scaling for Gauss–Legendre quadrature*:

$$\mathcal{E} \sim \frac{\pi}{4^n} \frac{f^{(2n)}(\xi)}{(2n)!} \tag{7.124}$$

The derivative term has been grouped in this way motivated by a standard Taylor series: as you will recall, the n-th term is divided by $n!$; we have therefore divided our $2n$-th derivative by $(2n)!$; this ratio is quite often bounded. If that is indeed the case, then the simple prefactor, $\pi/4^n$, gives us a practical criterion for the scaling of the error with n: as we increase n to $n+1$, we expect the error to decrease by a factor of roughly 4. This is a dramatically better scaling than what we had for Newton–Cotes methods: for example, the absolute error in the composite Simpson's rule was proportional to h^4, see Eq. (7.54), i.e., asymptotically $1/n^4$; compare that to the $1/4^n$ that we found in Eq. (7.124) for Gauss–Legendre quadrature. This is the difference between a power law and an exponential scaling. Thus, it should come as no surprise that Gaussian quadrature does a great job for many well-behaved functions.

[22] To prove this, use *Stirling's formula*, $n! \sim \sqrt{2\pi} e^{-n} n^{n+1/2}$.

Integrating from a to b

Up to this point we've only discussed Gauss–Legendre integration from -1 to 1, as per Eq. (7.89). We generally need to integrate from a to b. The solution is simply to carry out a change of variables:

$$t = \frac{b+a}{2} + \frac{b-a}{2}x \tag{7.125}$$

just like we did in our discussion of polynomial interpolation, see Eq. (6.13). We now use Eq. (7.125) to get the integration measure:

$$dt = \frac{b-a}{2}dx \tag{7.126}$$

At this point, we are able to recast our general task of computing $\int_a^b f(t)dt$ into a form that makes use of our earlier results on $\int_{-1}^1 f(x)dx$:

$$\int_a^b f(t)dt = \frac{b-a}{2}\int_{-1}^1 f\left(\frac{b+a}{2} + \frac{b-a}{2}x\right)dx \tag{7.127}$$

If we now employ Eq. (7.89) to go from integral to sum, this becomes:

$$\int_a^b f(t)dt \approx \frac{b-a}{2}\sum_{i=0}^{n-1} c_k f\left(\frac{b+a}{2} + \frac{b-a}{2}x_k\right) \tag{7.128}$$

Here the x_k and c_k are the standard quantities we derived in earlier sections.

Implementation

A Python implementation of Gauss–Legendre quadrature is given in Code 7.4. As before, we will compute the integral giving the electrostatic potential in Eq. (7.2); we import the integrand from `newtoncotes.py`. Similarly, we bring in the Legendre polynomial roots from `legroots.py` and derivatives from `legendre.py`. Observe that these two functions do all the heavy lifting; as a result, our code is very short.

The first function in this code produces the x_k's by finding the roots of $P_n(x)$ and the c_k's by using Eq. (7.117). Observe that `legroots()` returns a `numpy` array, which we now call `xs`. This is very convenient in the following line where, as usual, we don't need to employ any indices. Note also that we are calling `legendre()` passing in an array as the second argument: even though this function was written in chapter 2, before we had started using `numpy` arrays, it works seamlessly.

The next function, `gauleg()`, implements Eq. (7.128), taking us from x to t. This is intentionally written to have the same interface as our Newton–Cotes functions: we pass in the integrand, the integration limits, and n. As always, we carry out operations over entire arrays, so no indices are necessary here, either. The output of running this code is:

Code 7.4 gauleg.py

```
from legendre import legendre
from legroots import legroots
from newtoncotes import f
import numpy as np

def gauleg_params(n):
     xs = legroots(n)
     cs = 2/((1-xs**2)*legendre(n,xs)[1]**2)
     return xs, cs

def gauleg(f,a,b,n):
     xs, cs = gauleg_params(n)
     coeffp = 0.5*(b+a)
     coeffm = 0.5*(b-a)
     ts = coeffp + coeffm*xs
     contribs = cs*f(ts)
     return coeffm*np.sum(contribs)

if __name__ == '__main__':
     ans = np.log(1 + np.sqrt(2))
     print(ans)
     for n in range(2,10):
          print(n, gauleg(f,0.,1,n))
```

```
0.88137358702
2 0.881789806445
3 0.881331201938
4 0.881375223073
5 0.881373570699
6 0.881373584915
7 0.881373587172
8 0.881373587015
9 0.88137358702
```

The function we're integrating is non-polynomial, so we don't expect Gauss–Legendre to give the exact answer for the first few values of n. Overall, the results are quite impressive. Gauss–Legendre gives us the correct answer to three decimal digits using only $n = 2$ points and to four decimal digits with only $n = 3$ points. By the time we get up to $n = 9$ we have already determined the right answer for all the digits that are printed: this is

pretty spectacular. To compare, recall that Simpson's rule needed over 100 points to get to the same accuracy and even (non-wasteful) Romberg integration required 33 points. Of course, as our discussion of the Gauss–Legendre error formula showed, the convergence to the right answer is always impacted by specific properties of the function we choose to integrate.

On the other hand, it is worth pointing out that each value of n necessitates completely different values of the abscissas x_k. This is not a problem if your $f(x)$ is a simple function, as in the present example. However, you can imagine that in practical applications computing $f(x)$ may be rather costly. In one of the problems, you are guided toward implementing a non-wasteful adaptive trapezoid integrator; something analogous *cannot* be straightforwardly done for Gauss–Legendre integration. There are schemes, most notably *Gauss–Kronrod quadrature*, where the Gauss–Legendre points are kept and accompanied by new ones. Another option is *Clenshaw–Curtis quadrature*, which expands the integrand using Chebyshev polynomials and can therefore compute the integration weights very efficiently, via the FFT; this method's abscissas obey *nesting*, so they can be re-used as you increase n. Even so, the plain Gauss–Legendre approach will be enough for us. From a practical perspective, you can start with, say, $n = 10$ and then increase n by steps of your choosing, to empirically test the convergence for your problem.

7.5.3 Other Gaussian Quadratures

In the previous sections, we focused on Gauss–*Legendre* quadrature only. This was partly because that's the more general and widely employed technique and partly for pedagogical reasons: instead of giving a laundry list of formulas for many different cases, we studied one case carefully, deriving everything by ourselves. In the present short section, we will provide a qualitative argument of how one goes about generalizing things; some of the problems investigate these new formulas further.[23]

The general form of the problem at hand is:

$$\int_a^b w(x)f(x)dx \approx \sum_{i=0}^{n-1} c_k f(x_k) \tag{7.129}$$

where $w(x)$ is a non-negative *weight function*. As you can see, the presence of $w(x)$ on the left-hand side (only) is new. The weight function can be singular, but it needs to be integrable; crucially, it can be non-polynomial. As before, the Gaussian quadrature approach allows you to find the exact answer when $f(x)$ is a polynomial of degree up to $2n - 1$: in that case, the full integrand does *not* need to be a polynomial. Here are three standard

[23] As usual, our discussion in the main text is carried out *pros ten chreian ikanos*, i.e., sufficiently as to need, to quote Aristotle (*Nicomachean Ethics*, 1133b); the alternative would be for us to continue *ad taedium*.

choices for the weight function, which have been worked out in the literature:

$$\begin{aligned} &\text{Gauss–Chebyshev: } w(x) = \frac{1}{\sqrt{1-x^2}} \\ &\text{Gauss–Laguerre: } w(x) = x^\alpha e^{-x} \\ &\text{Gauss–Hermite: } w(x) = e^{-x^2} \end{aligned} \tag{7.130}$$

Gauss–Legendre is equivalent to taking $w(x) = 1$, so a possible (but non-ideal) strategy for avoiding other Gaussian quadratures is to treat the entire integrand $w(x)f(x)$ as what used to be called your $f(x)$ and then apply Eq. (7.128).

A better line of reasoning goes as follows: once again, start from the Hermite interpolation formula of Eq. (7.95). This time, multiply with $w(x)$ before integrating over the general interval $[a, b]$, to find:

$$\begin{aligned} \int_a^b w(x)f(x)dx &= \int_a^b w(x)p(x)dx \\ &= \sum_{k=0}^{n-1} f(x_k)\int_a^b w(x)\alpha_k(x)dx + \sum_{k=0}^{n-1} f'(x_k)\int_a^b w(x)\beta_k(x)dx \\ &= \sum_{k=0}^{n-1} c_k f(x_k) + \sum_{k=0}^{n-1} d_k f'(x_k) \end{aligned} \tag{7.131}$$

In the last step we defined:

$$c_k = \int_a^b w(x)\alpha_k(x)dx, \qquad d_k = \int_a^b w(x)\beta_k(x)dx \tag{7.132}$$

As before, we will demand that the d_k's vanish. These are:

$$d_k = \int_a^b w(x)\beta_k(x)dx = \int_a^b w(x)(x-x_k)L_k^2(x)dx = \frac{1}{L'(x_k)}\int_a^b w(x)L(x)L_k(x)dx \tag{7.133}$$

Once again, $L(x)$ is a monic polynomial of degree n and $L_k(x)$ is a polynomial of degree $n-1$. In Eq. (7.98) we ensured that this integral vanishes by taking the nodes, x_k, to be the roots of a Legendre polynomial. We did this because, for $w(x) = 1$ and in the interval $[-1, +1]$, it is precisely the Legendre polynomials that obey such an orthogonality criterion.

Our more general line of thinking here now leads us to the roots of polynomials that are *w-orthogonal in the interval from a to b*. These turn out to be Chebyshev polynomials, Laguerre polynomials, and Hermite polynomials for the cases listed in Eq. (7.130); the corresponding standard intervals are $[-1, 1]$, $[0, \infty)$, and $(-\infty, \infty)$, respectively. Incidentally, you should make sure not to get confused by the presence of the name *Hermite* in Gauss–Hermite: we have derived *all* Gaussian quadratures using Hermite *interpolation*. What's special about Gauss–Hermite quadrature is that it involves Hermite polynomials.[24]

We don't go into specifics, but it should be relatively obvious that the weights can now be computed by a generalization of Eq. (7.105), possibly augmented by a Christoffel–Darboux identity for the specific set of polynomials, as well as the corresponding normalization

[24] These were implemented in section 3.5 and you were asked to find their roots in a problem in chapter 5.

and recurrence relation. Similarly, the error scaling will follow from the generalization of Eq. (7.121) to include a $w(x)$ in the integrand. In broad strokes, that's all there is to it.

7.6 Complicating the Narrative

Most statements about errors that we've made in this chapter so far assumed we were dealing with well-behaved Taylor series, with each term being smaller in magnitude than the previous one. In practice, this is equivalent to saying that our function can be reasonably well approximated locally by a polynomial. However, there are many cases, even non-pathological ones, which cannot be effectively captured by polynomial behavior. For example, as Forman Acton liked to point out, polynomials don't have asymptotes [2]. In what follows, we go over some cases where the error budget behaves differently than what we saw above, we discuss analytical manipulations one can carry out *before* computing an integral, discuss the problem of integrating over more than one variable, and close with some advice on which integration approach(es) you should prefer.

7.6.1 Periodic Functions

Given the ability to use Gaussian quadrature or even Simpson's rule, you may be tempted to say that a method as simple as the trapezoid rule, with its $O(h^2)$ approximation error as per Eq. (7.37), should *never* be employed. As you'll see when you solve the relevant problem, this conclusion is wrong; the problem studies the function $f(x) = e^{\sin(2x)}$, which we've repeatedly encountered in earlier chapters, but the argument explaining what's going on is much more general.

As we saw when introducing the Romberg recipe, our conclusion for the trapezoid rule was merely that the *leading* error is $O(h^2)$; in Eq. (7.70) we saw that the full expression contains many more (even) degrees:

$$\mathcal{E} = c_2 h^2 + c_4 h^4 + c_6 h^6 + \cdots \tag{7.134}$$

This fact, namely that only even degrees survive, was all we needed to employ Richardson extrapolation repeatedly. This time around, we are interested in investigating the error budget in more detail: Eq. (7.70) was actually merely a way of condensing our result employing the Euler–Maclaurin summation formula; this was Eq. (7.41), which explicitly listed the coefficients of the powers of h:

$$\begin{aligned}\mathcal{E} = &-\frac{h^2}{12}\left[f'(b) - f'(a)\right] + \frac{h^4}{720}\left[f'''(b) - f'''(a)\right] \\ &- \frac{h^6}{30\,240}\left[f^{(5)}(b) - f^{(5)}(a)\right] + \frac{h^8}{1\,209\,600}\left[f^{(7)}(b) - f^{(7)}(a)\right] + \cdots\end{aligned} \tag{7.135}$$

These coefficients involve Bernoulli numbers, factorials and, crucially, *the differences between odd-order derivatives at the endpoints*.

If you take a periodic function and integrate it over one full period (or more periods), you will have $f'(a) = f'(b)$, $f'''(a) = f'''(b)$, and so on; thus, the error estimate from

Eq. (7.135) will have a zero coefficient for each power.[25] This means that in such a case the composite trapezoid rule will have a *dramatically* better error behavior than what you usually find; as you will discover in the aforementioned problem, in some cases it even outperforms Gauss–Legendre quadrature, whose praises we were singing in the previous section. Of course, all these benefits vanish the second you decide to integrate a periodic function over an interval which is *not* a period.

7.6.2 Singularities

A major complication arises when you are faced with an integrand that has a singularity, or a singular derivative at one of the endpoints. If the singularity is not at an endpoint, you can slice up your interval so that the singularity is at an endpoint. We will discuss five examples here to give you a flavor of the techniques involved; the problems invite you to apply analogous tricks to other integrals.

Singularity at the Left Endpoint

Examine the integral:

$$I_A = \int_0^2 \frac{\sin x}{\sqrt{x}} dx \tag{7.136}$$

This has a singularity[26] at the left endpoint, $x = 0$, so closed integration methods (e.g., the trapezoid rule) naively applied to this problem will give "not a number" (along with a `RuntimeWarning`) or a `ZeroDivisionError`; furthermore, if you plot the integrand you will see that it is vertical near the origin.

The solution is to make a change of variables; taking $u = \sqrt{x}$ leads to:

$$I_A = \int_0^{\sqrt{2}} 2\sin(u^2) du \tag{7.137}$$

This has no singularities and no vertical slopes, so closed methods will work just fine.

Discontinuous Derivative

Now take the following integral:

$$I_B = \int_0^2 \sqrt{x}\sin x dx \tag{7.138}$$

This time, the integrand is not singular, but its derivative(s) will be singular. A problem asks you to apply the trapezoid rule to a similar integral; there you will see in action that the error does not behave according to Eq. (7.134), i.e., it is much worse. To see why, recall that Eq. (7.37) told us that the leading error for the trapezoid rule is not simply $O(h^2)$, but actually $-h^2 f''(\xi)(b-a)/12$. This means that you get in trouble if you are dealing with non-smooth functions. We will now see three different ways of addressing this situation.

[25] The only possible exception being the remainder term that was dropped from Eq. (7.39).

[26] The integrand has a *numerical* singularity, since floats don't know about L'Hôpital's rule.

First, we can make the same change of variables as above, namely $u = \sqrt{x}$. This gives:

$$I_B = \int_0^{\sqrt{2}} 2u^2 \sin(u^2) du \tag{7.139}$$

Our new integrand is arbitrarily often differentiable in our interval.

Second, you can choose to slice up the integral:

$$I_B = \int_0^{0.01} \sqrt{x} \sin x dx + \int_{0.01}^{2} \sqrt{x} \sin x dx \tag{7.140}$$

where the 0.01 is, obviously, not set in stone but merely indicative of a "small" value. Notice that for the second term, whose endpoints do not include the origin, the integrand is arbitrarily often differentiable. For the first integral, since it only involves small values of x, you are allowed to Taylor expand $\sin x$ in it:

$$\int_0^{0.01} \sqrt{x} \sin x dx = \int_0^{0.01} \sqrt{x} \left(x - \frac{x^3}{6} + \frac{x^5}{120} - \cdots \right) dx \tag{7.141}$$

Each of these integrals can now be carried out analytically.

Third, we often try to *subtract off the singularity*, which in our case translates to subtracting off the singularity in the derivative. Practically speaking, what this means is we add in and subtract out another function that can be analytically integrated *and* exhibits the same misbehavior as our integrand. In our case, this can be done as follows:

$$I_B = \int_0^2 \sqrt{x} \sin x dx = \int_0^2 \sqrt{x}(\sin x - x) dx + \int_0^2 \sqrt{x} x dx = \int_0^2 \sqrt{x}(\sin x - x) dx + \frac{8\sqrt{2}}{5} \tag{7.142}$$

The integrand we are left with in the last step has a continuous third derivative, so an approach like the trapezoid rule won't have as hard a time tackling it.[27]

Singularity at the Right Endpoint

Examine the following integral:

$$I_C = \int_0^1 \frac{\sin x}{\sqrt{1-x}} dx \tag{7.143}$$

This one has a singularity at the right endpoint. The trick is to move the singularity to the origin, by using the change of variables $u = 1 - x$, which leads to:

$$I_C = \int_0^1 \frac{\sin(1-u)}{\sqrt{u}} du \tag{7.144}$$

where the minus signs from the differential and the new limits of integration have balanced each other out. We now have the same problem with a square root at the origin as in I_A, so we will make a further change of variables, namely $v = \sqrt{u}$, leading to:

$$I_C = \int_0^1 2 \sin(1 - v^2) dv \tag{7.145}$$

which has no issues with singularities or smoothness.

[27] Of course, as you discovered in chapter 2, $\sin x - x$ suffers from subtractive cancellation for small x, so you would benefit from employing a Taylor expansion here.

Singularities at Both Endpoints

We now turn to a slightly different integral:

$$I_D = \int_0^1 \frac{\sin x}{\sqrt{x(1-x)}} dx \tag{7.146}$$

This looks similar to I_C, but now has problems at both endpoints. The trick is to split it into two integrals by hand:

$$I_D = \int_0^{1/2} \frac{\sin x}{\sqrt{x(1-x)}} dx + \int_{1/2}^1 \frac{\sin x}{\sqrt{x(1-x)}} dx \tag{7.147}$$

where the matching point was arbitrarily chosen to be the midpoint. Now the first integrand has a singularity at the left endpoint and the second integrand a singularity at the right endpoint. The first integral should be tackled by saying $u = \sqrt{x}$ and the second integral by taking $u = 1 - x$ followed by $v = \sqrt{u}$.

Again, Singularities at Both Endpoints

As our final example we look at another integral that raises concerns at both endpoints:

$$I_E = \int_{-1}^1 \frac{\sin x}{\sqrt{1-x^2}} dx \tag{7.148}$$

This time taking $u = \sqrt{x}$ complicates things at the left endpoint. What we'll do, instead, is to make the change of variables $u = \sqrt{1-x}$, which leads to:

$$I_E = \int_0^{\sqrt{2}} 2\frac{\sin(1-u^2)}{\sqrt{2-u^2}} du \tag{7.149}$$

The denominator may look similar to the one we started from, but now it only has an issue at the right endpoint.

You could now imagine employing yet another trick, most likely chosen from the list of those we've already encountered. Instead, we recognize that I_E is an example of a standard Gaussian quadrature, namely *Gauss–Chebyshev*, as per Eq. (7.130); the weights have already been studied and tabulated, so this integral can be straightforwardly evaluated. The lesson of this story is to always examine your integrals to see if they are of standard form *before* you start using the tricks of the trade.

7.6.3 Infinite Intervals

We now turn to another scenario, one where an infinity appears not in the integrand but in the endpoints of integration; as you may recall, these are called *improper integrals*.

One Infinity

Here's a straightforward example:

$$I_F = \int_1^\infty \frac{e^{-x}}{x+1} dx \tag{7.150}$$

If you wanted to use a standard (Newton–Cotes or Gaussian) quadrature approach, you would be faced with the question of how to handle the right endpoint. One possible solution is to take it to be large but finite, say, 5 or 10; but is that enough? You could try increasing it further to, say, 20 or 50 and see if the result of numerical integration keeps changing. The problem is that once you do that, you need to also make sure you employ more integration abscissas. But a simple plot of this integrand shows you that its values are tiny when x is above 10 or so. This means that you would be employing the majority of the abscissas to integrate a tiny tail of the function.

In the spirit of the previous section, what we do, instead, is a change of variables; taking $u = 1/x$ our integral becomes:

$$I_F = \int_0^1 \frac{e^{-1/u}}{u(u+1)} du \tag{7.151}$$

Note how the new integration interval is *not* infinite. Of course, now there is a new singularity at the origin, but that can be tackled using the techniques of the previous section.

Another Infinity

Of course, the trick we just introduced worked only because one of the endpoints was infinite *and* the other endpoint was non-zero; you can immediately see that if we had been faced with, say:

$$I_G = \int_0^\infty \frac{1}{\sqrt{x^8 + x}} dx \tag{7.152}$$

then the change of variables $u = 1/x$ would still lead to an infinite endpoint. One approach is to manually slice this integral up:

$$I_G = \int_0^1 \frac{1}{\sqrt{x^8 + x}} dx + \int_1^\infty \frac{1}{\sqrt{x^8 + x}} dx \tag{7.153}$$

The first integral is over a finite interval and the second one is amenable to the change of variables $u = 1/x$ and will, crucially, have no singularities. The former integral doesn't have an infinite endpoint, but it does have a singularity at the origin; as in the previous subsection, we can take $u = \sqrt{x}$ to eliminate that.

Another approach involves a single transformation for the entire interval. This one is sufficiently important that it deserves to be emphasized:[28]

$$u = \frac{1}{1+x} \tag{7.154}$$

It is straighforward to see that this leads to a finite interval (0 gets mapped onto 1 and ∞ onto 0). Similarly, it can be solved to give $x = (1-u)/u$ and $dx = -du/u^2$. Using this transformation, I_G turns into:

$$I_G = \int_0^1 \frac{u^2}{\sqrt{(1-u)^8 + u^7(1-u)}} du \tag{7.155}$$

[28] Of course, this is not unique; for example, $u = x/(1+x)$ is another choice.

This has taken care of the infinite interval, but is suffering from a singularity at the right endpoint. However, the sequence of steps discussed earlier can cure all issues here, as you will discover in a problem.

Two Infinities

Sometimes we are faced with an integration from minus infinity to plus infinity:

$$I_H = \int_{-\infty}^{\infty} \frac{e^{-x^2}}{\sqrt{x^2+1}} dx \tag{7.156}$$

What's typically done in these cases is to slice up the integral into two pieces, one going from $-\infty$ to 0 and the other one from 0 to $+\infty$. In the present case, this allows us to cut the work in half, since the integrand is even, but the technique is of wider applicability.

Yet another approach is to recognize that I_H is an example of a standard Gaussian quadrature, namely *Gauss–Hermite*, as per Eq. (7.130); that means that the weights have already been studied and tabulated, so this integral can be straightforwardly evaluated. This is a general feature: when faced with integrals from $-\infty$ to $+\infty$ you should always consider Gauss–Hermite and, similarly, for integrals from 0 to ∞ (such as I_G) you should think of Gauss–Laguerre.

7.6.4 Multidimensional Integrals

It is, in principle, straightforward to generalize our earlier discussion to the case of multidimensional problems. The easiest way to do so is to use *Fubini's theorem*, whereby a multidimensional integral is written as a sequence of iterated one-dimensional integrals.

To be specific, let's look at the following two-dimensional integral:

$$I = \int_{-1}^{1} \int_{-1}^{1} f(x,y) dx dy \tag{7.157}$$

We can re-express this by first defining:

$$F(y) = \int_{-1}^{1} f(x,y) dx \tag{7.158}$$

This keeps y fixed and carries out the integration over x. Then, we can use this new entity to rewrite our original integral as follows:

$$I = \int_{-1}^{1} F(y) dy \tag{7.159}$$

which is also a one-dimensional integral.

Numerically, we can use either Newton–Cotes or Gauss–Legendre quadrature, both of which take the form Eq. (7.7), to express the above steps as follows:

$$F(y) \approx \sum_{i=0}^{n-1} c_i f(x_i, y) \tag{7.160}$$

and then:

$$I \approx \sum_{j=0}^{n-1} c_j F(y_j) \tag{7.161}$$

Putting the last two equations together leads to:

$$I \approx \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} c_i c_j f(x_i, y_j) \tag{7.162}$$

Implementing this should be straightforward. A problem asks you to employ Gaussian quadrature for a five-dimensional integral: you simply generalize Eq. (7.162) to the five-variable case, which also means you will employ five sets of abscissas and weights.

As you will learn when you attempt that problem, as you increase the number of dimensions, you are faced with the following issue: should you keep the number of points in one dimension fixed, or the total number of points fixed? Let's be specific: assume d is the number of dimensions, n is the number of points in each dimension (just like in our one-dimensional studies earlier in this chapter), and $\mathcal{N} = n^d$ is the total number of points across all dimensions. As is shown in Eq. (7.162), this $\mathcal{N}$ gives the total number of function evaluations you will need to perform.[29] As we've noted repeatedly throughout this volume, in the real world it is precisely a function evaluation that is the slowest part of a computation: this is typically the result of separately running another lengthy calculation.

In practice, one is limited by "total wall-clock time"; this means that we are interested in determining the scaling of a given approach with respect to the total number of function evaluations, $\mathcal{N}$. For all the (one-dimensional) quadrature methods we've encountered, we can describe the leading error as $\mathcal{O}(h^p)$, similarly to what we saw in section 7.4. Using the definition of h from Eq. (7.9), we can express this leading error as $\mathcal{O}(n^{-p})$. Note that the first use of $\mathcal{O}$ applies to small h, whereas when we switch to n we are interested in large n. Of course, this is the error we make in one dimension, so in d dimensions we should probably multiply it with d; however, if we are interested in examining the scaling with n (or $\mathcal{N}$), then d can be considered a constant prefactor which is therefore omitted. If we now take our leading error, $\mathcal{O}(n^{-p})$, and express it in terms of the total number of function evaluations, $\mathcal{N} = n^d$, we arrive at the following expression for the *total error in d-dimensional integration*:

$$\mathcal{E} = \mathcal{O}\left(\mathcal{N}^{-p/d}\right) \tag{7.163}$$

To reiterate, p is the power of the leading error in the quadrature rule we're employing ($\mathcal{O}(h^p)$), d is the number of dimensions, and $\mathcal{N}$ is the number of function evaluations. When d is large, this scaling is very poor; for example, take Simpson's rule, $p = 4$, applied

[29] In the present chapter, n denotes the number of abscissas and N the number of panels; asymptotically, these two are equal: $\mathcal{O}(n) = \mathcal{O}(N)$. We are using a new symbol for the total number of function evaluations, $\mathcal{N}$.

to a 100-dimensional integral: the scaling is $O(N^{-1/25})$, so as we double the total work, i.e., double N, the error is divided by $2^{1/25} \approx 1.03$. This is known as the *curse of dimensionality*: when d is large, doubling our effort is basically a waste of time.

This curse of dimensionality provides a natural entry point to section 7.7, which studies an approach to integration known as *Monte Carlo*; as you will see there, Monte Carlo has an error that scales as $O(N^{-1/2})$. Crucially, Monte Carlo exhibits this scaling *regardless of the dimensionality!* To return to our example of a 100-dimensional integral: in Monte Carlo when you double the total work the error is divided by $\sqrt{2} \approx 1.41$. This means that the effort you put in toward doubling N actually pays off. In practice, for one (or a few) dimension(s) it is standard to employ conventional quadrature methods like those discussed above, but for anything more complicated one has to choose a Monte Carlo algorithm. Before we see how that all works, let's do a brief recap.

7.6.5 Evaluating Different Integration Methods

We have encountered several quadrature techniques and seen their strengths and weaknesses. Here we summarize these comments:

- There is no substitute for plotting your integrand and thinking about it before you numerically integrate.
- If you can use an analytical trick to eliminate a singularity or a singularity in the derivative or an infinite interval, you should do so ahead of time.
- If your integrand is periodic and you are integrating over a full period (or several periods), the composite trapezoid rule can do a great job.
- If your (proper or improper) integral is of a standard Gaussian quadrature form, you should use the tabulated nodes and weights.
- For most smooth integrands where you need high accuracy, Romberg integration or Gauss–Legendre integration are the go-to solutions.
- If you need your abscissas to nest, employ a Newton–Cotes rule (or Clenshaw–Curtis quadrature).
- If you're not too worried about the accuracy but need a dependable error[30] (or if you are externally constrained to use equally spaced nodes), you should employ an adaptive Simpson's rule; if you have an even number of points, remove one point and use the trapezoid rule on the last panel.
- If you're dealing with a multidimensional integral, use Gaussian quadrature in a few dimensions and Monte Carlo in many dimensions.

7.7 Monte Carlo

The section's title comes from the casino in Monaco and reflects the "chance" (or random) aspects of such approaches. Monte Carlo techniques employ random numbers to tackle

[30] Of course, the dependability of the error rests on a well-behaved Taylor expansion, i.e., a smooth integrand.

either naturally stochastic[31] processes or nonprobabilistic problems. In keeping with the theme of the present chapter, we will focus on the latter, namely numerical integration. We start by discussing what "random" numbers are and how to produce them; we then turn to a detailed discussion of one-dimensional Monte Carlo quadrature, before addressing the real-world problem of multidimensional integration.

7.7.1 Random Numbers

Determining the exact meaning of "random" can become complicated. A nice example of patterns that we notice which may (or may not) really be there has to do with the decimal digits of the number π. For most practical purposes, the digits of π are random. However, when looking at two-digit repeats, we notice that the first such pair to re-appear is 26:

$$3.14159\overset{\sqcap}{2}6535897932384\overset{\sqcap}{2}64338327950 \qquad (7.164)$$
$$\sqcup\,\sqcup\,\sqcup \qquad\qquad \sqcup\,\sqcup\,\sqcup$$

We also notice that the second occurrence of 26 shows up in the middle of a strange repetition pattern: 79, 32, and 38 are repeated in reverse order contiguously, as 38, 32, and 79. The moral of this story is that checking for patterns/correlations can easily devolve into numerology. This is why one needs quantitative tests of the randomness of a given sequence. We will only scratch the surface of this topic here.

Computers produce what are known as *pseudorandom numbers*: the use of the modifier "pseudo" is due to the fact that computers are (supposed to be) deterministic systems. Thus, they produce sequences where each number is completely determined by its predecessor(s). However, if someone who does not have access to the random-number generation algorithm is led to believe that the sequence is truly random, then we have a "good" random-number generator. Thus, when dealing with good pseudorandom number sequences we tend to simply drop the "pseudo" and speak simply of random-number sequences.

Linear Congruential Generator

We start with a simple approach that can produce reasonably good sequences of uniformly distributed random numbers, namely a *linear congruential generator*. While high-quality libraries typically employ more complicated algorithms, what we discuss here should be enough to give you a flavor of what's involved.

Integers

We first see how to produce randomly distributed integers and turn to floating-point numbers next. A linear congruential generator (LCG) produces integers from 0 to $m-1$, where m is an integer appearing in the formula:

[31] The word comes from the Greek for "aim" or "guess".

$$u_i = (au_{i-1} + c) \bmod m \tag{7.165}$$

Here a and c are also integers. The way this works is that (for a fixed choice of a, c, and m) we start with an integer u_{i-1} and this equation instructs us how to construct the next integer in the sequence, u_i. This sequence has to start somewhere: we call the first/0th number (the one that jumpstarts the generation of numbers) the *seed* and denote it by u_0.

As an example, assume $a = 4$, $c = 1$, and $m = 15$ and pick the seed to be $u_0 = 5$:

$$\begin{aligned} &u_0 = 5, \quad u_1 = (4 \times 5 + 1) \bmod 15 = 6, \quad u_2 = (4 \times 6 + 1) \bmod 15 = 10, \\ &u_3 = (4 \times 10 + 1) \bmod 15 = 11, \quad u_4 = (4 \times 11 + 1) \bmod 15 = 0, \\ &u_5 = (4 \times 0 + 1) \bmod 15 = 1, \quad u_6 = (4 \times 1 + 1) \bmod 15 = 5 \end{aligned} \tag{7.166}$$

We have stopped here, because we know that as soon as we get a repeated number (5 in this case) all the numbers to follow will be obeying the same trend: this is a deterministic algorithm that will keep repeating the same sequence over and over again. The distinct numbers are those from u_0 to u_5, so we say we have a *period* of 6. To be explicit, the parameters we used here lead to the sequence:

$$5, 6, 10, 11, 0, 1, 5, \ldots \tag{7.167}$$

which is infinitely repeating. Obviously, this is not good enough for practical purposes.

Note that, in our example above, we divided with $m = 15$ but got a period of 6. It's fairly easy to see that an LCG sequence cannot have a period that is larger than m: since we're taking mod m, we're producing numbers from 0 to $m - 1$. But there only exist m distinct numbers from 0 to $m - 1$: the minute we get a repeat, we know the entire sequence will repeat after that.[32] Given that a fixed m leads to numbers from 0 to $m - 1$ and can therefore have a maximum period of m, it stands to reason that we would like to have m be large; it is standard to pick m to be 2 raised to a large power. However, simply increasing m is not enough: it's generally a good idea to also pick a to be large. Of course, using a large m and a large a only addresses the question of the large period, not that of the "quality" of our random number sequence; we return to this question below.

So far, we've been discussing Eq. (7.165): once the parameters a, c, and m are set, this formula only requires knowledge of the current number in order to produce the next one. However, one could concoct more complicated generators, which depend on a larger part of the sequence's history, for example:

$$u_i = \left(au_{i-j} + cu_{i-k}\right) \bmod m \tag{7.168}$$

for fixed values of j and k. In this case, to jumpstart the generation of numbers one would have to start not with one number (the seed), but with a sequence of such numbers.

[32] Note that this holds for the LCG algorithm, not for random sequences in general.

Floats

In applications, we often need not integer random numbers, but floats/real numbers. The most common variety in which these are needed is (uniformly distributed) random numbers from 0 to 1. It's easy to see how to produce these numbers starting from a sequence of integers generated using Eq. (7.165); simply divide with m:

$$r_i = \frac{u_i}{m} \tag{7.169}$$

This automatically ensures that the output floats will be from 0 (inclusive) to 1 (exclusive). Since our integers go from 0 to $m-1$, we'll never get an r_i that is all the way up to 1 itself, but that's generally not a problem, for very large values of m.

To be explicit, the sequence of integers in Eq. (7.167) will (upon division with $m = 15$) lead to the sequence of roughly:

$$0.3333, 0.4, 0.6667, 0.7333, 0.0, 0.0667, 0.3333, \ldots \tag{7.170}$$

This is still a random-number sequence (this time, of floats) of period 6.

In some applications we need random floats not in $[0, 1)$ but in $[a, b)$. Again, it's easy to do this. Starting from r_i in $[0, 1)$, we transform as follows:

$$x_i = a + (b - a)r_i \tag{7.171}$$

where x_i is now in $[a, b)$. We can use this transformation when we need floats from -1 to 1: the relation above gives us $x_i = -1 + 2r_i$, which can be re-expressed as $x_i = 2(r_i - 0.5)$.

Tests of Random Number Generators

In the preceding discussion, we saw that pseudorandom number generators eventually repeat, so one's goal should be to have as large a period as possible. However, one should also consider if the random-number generator has good statistical quality; this basically means that there should not be glaring *correlations* along the sequence of random numbers. For example, the sequence:

$$1, 2, 3, 4, 5, 6, \ldots \tag{7.172}$$

is clearly not random, since you can easily predict the next number in the sequence (i.e., you know that if your current number is 37 then your next number will be 38). Here's another example:

$$u_0, 101 - u_0, u_1, 101 - u_1, u_2, 101 - u_2, \ldots \tag{7.173}$$

This is also not a very random sequence: very often, if you know one number you can predict the next one. For example, if you're at the start of a new pair and the number 33 appears, then you know that the next number will be 68. This sequence contains strong

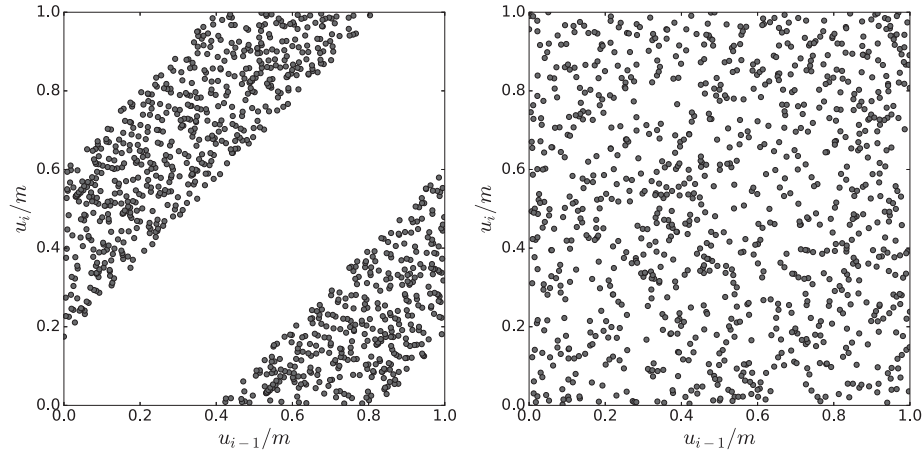


Fig. 7.5 Random numbers plotted pairwise for a "bad" (left) and a "good" (right) generator

pairwise correlations. Both these examples are hinting at what a random-number sequence is by exhibiting its opposite: non-random number sequences, which are often predictable (at least in part).

One way of testing for correlations is by plotting pairs of numbers, $(x_i, y_i) = (u_{i-1}, u_i)$. If such a plot exhibits regularity, something is wrong. The left panel of Fig. 7.5 shows the result of generating 1000 uniformly distributed random numbers from the generator:

$$u_i = \left[(2^{18} + 1)u_{i-1} + 7\right] \bmod 2^{32} \tag{7.174}$$

and plotting them pairwise; the right panel employs a generator that is almost identical: the only change is in the coefficient, $a = 1\,812\,433\,253$. In both cases we started from the seed $u_0 = 314\,159$ and have divided the u's with 2^{32}. The left panel exhibits a clear pattern: all the pairs end up in one of two bands; in a problem, you are asked to reproduce these plots using Python. Of course, this test involves "visual inspection", so it is still prone to human error (e.g., is it obvious that the right panel is exhibiting no patterns whatsoever?); in reality, this is merely the visual component of a systematic approach known as the *serial test*, which checks tuples of successive numbers to see if they are independent of each other. Another problem explores the *equidistribution test*: you are asked to divide up the region from 0 to $m - 1$ into "bins" and then count how many numbers fall into each bin.

Even though up to this point we've been producing (or trying to produce) uniformly distributed random numbers, it's important to realize that *random* is not the same thing as *uniform*. Random in the sense that we're using it here means that there are few/no correlations and that there is a very large period. If you're dealing with numbers from, say, 1 to 100, "random" does not necessarily mean that all numbers from 1 to 100 are equally likely to occur. For example, we can have a random number sequence that follows a normal/Gaussian distribution centered around 50: that means that numbers around 50 are much more likely to occur than, say, numbers around 10 or 90, which are close to the tails of the distribution. Such numbers are sometimes called *normal deviates*, in contradistinction

to random numbers that are uniformly distributed, which are called *uniform deviates*. We discuss how to produce non-uniformly distributed random numbers in later sections.

Random Numbers in Python

For industrial-scale applications, we typically do not rely on our own random-number generators. Python itself includes a high-quality generator in the `random` module:

```
>>> import random
>>> random.seed(314159)
>>> random.random()
0.19236379321481523
>>> random.random()
0.2868424512347926
```

We first call the `random.seed()` function to provide the seed. We then see that repeated invocations to the `random.random()` function lead to new random numbers, uniformly distributed in $[0, 1)$. Python's `random.random()` uses a Mersenne Twister algorithm which has very good statistical properties and a period of $2^{19,937} - 1$, which is large enough for most practical applications.[33] You will compare its properties to LCG ones in a problem.

If you need several random numbers stored in an array, you could hand-roll a solution using `random.random()`. However, it's probably best to directly employ the functionality contained in `numpy.random.uniform()`; this function takes in three parameters: the first two determine the interval $[a, b)$ while the third one contains the shape of the output array:

```
>>> import numpy as np
>>> np.random.seed(314159)
>>> np.random.uniform(-1,1,4)
array([ 0.63584662, 0.10209259, -0.16044929, -0.80261629])
>>> np.random.uniform(-1,1,(2,3))
array([[ 0.6220415 , 0.93471281, -0.80358661],
       [ 0.60372074, 0.20980423, 0.16953762]])
```

where we also showed the seeding taking place through a call to `np.random.seed()`.[34]

7.7.2 Monte Carlo Quadrature

We now turn to the question of how random numbers can be used to compute integrals, starting from the one-dimensional case for simplicity. At this point, you might want to have a look at our brief recap on probability theory in Appendix C.3.

[33] Such a number is known as a Mersenne prime; this explains the presence of Descartes' chief correspondent in the name of an algorithm developed in 1997.

[34] While `np.random.seed()` is fine for our limited needs here, if you think you may end up needing independent streams in the future you should use `np.random.RandomState()` instead.

Population Mean and Population Variance

Our starting point will be Eq. (C.31), giving us the *expectation* (which is an integral) of a function f of a continuous random variable X:

$$\langle f(X)\rangle = \int_{-\infty}^{+\infty} p(x)f(x)dx \tag{7.175}$$

For now, we take $p(x)$, the probability density function, to be uniform from a to b, with the value $p(x) = 1/(b-a)$, and zero elsewhere[35] (we'll generalize this in section 7.7.3 below). This leads to the following result:

$$\langle f(X)\rangle = \frac{1}{b-a}\int_a^b f(x)dx \tag{7.176}$$

Except for the $1/(b-a)$ prefactor, the right-hand side is exactly of the form of Eq. (7.7), which is the problem we've been solving this entire chapter. To keep the terminology straight, we will call this $\langle f(X)\rangle$ the *population mean*. Similarly, we can take Eq. (C.32), giving us the *variance* of a function of a random variable, in terms of an integral:

$$\text{var}[f(X)] = \langle [f(X) - \langle f(X)\rangle]^2\rangle = \langle f^2(X)\rangle - \langle f(X)\rangle^2 \tag{7.177}$$

and specialize to the case where $p(x) = 1/(b-a)$, to find:

$$\text{var}[f(X)] = \frac{1}{b-a}\int_a^b f^2(x)dx - \left(\frac{1}{b-a}\int_a^b f(x)dx\right)^2 \tag{7.178}$$

To keep the terminology consistent, we call this the *population variance*. Obviously, its square root gives us the *population standard deviation*, $\sqrt{\text{var}[f(X)]}$.

Crucially, both the population mean and the population variance, $\langle f(X)\rangle$ and $\text{var}[f(X)]$, are unknown, i.e., we don't know the value of either $\int_a^b f(x)dx$ or $\int_a^b f^2(x)dx$, which appear in Eq. (7.176) and Eq. (7.178); this is precisely why we wish to employ Monte Carlo integration. In what follows, we will learn how to *estimate* both $\langle f(X)\rangle$ and $\text{var}[f(X)]$.

Sample Mean and Its Variance

Assume that the random variables $X_0, X_1, X_2, \ldots, X_{n-1}$ are drawn randomly from $p(x)$, which for us is uniform from a to b. For each of these random variables X_i, we act with the function f, leading to n new random variables, $f(X_0), f(X_1), f(X_2), \ldots, f(X_{n-1})$. (Remember, a function of a random variable is another random variable.) Similarly, the *arithmetic average* of several random variables is also a random variable. We define:

[35] We do this to ensure that the probability density is normalized: $\int_{-\infty}^{+\infty} p(x)dx = \int_a^b 1/(b-a)dx = 1$. If we were interested in an improper integral, we would make a different choice here.

$$\overline{f} = \frac{1}{n}\sum_{i=0}^{n-1} f(X_i) \tag{7.179}$$

We call this the *sample mean*: it is the arithmetic average of the function value over n samples. Crucially, this $\overline{f}$ is a quantity computed from a finite number of samples, n; this means it is quite different from the population mean, $\langle f(X)\rangle$, which as we see in Eq. (7.176) is an integral over continuous values of x.

We will now see that the sample mean, $\overline{f}$, can be used to *estimate* the population mean, $\langle f(X)\rangle$. Let us examine the expectation of the sample mean:[36]

$$\langle\overline{f}\rangle = \langle\frac{1}{n}\sum_{i=0}^{n-1} f(X_i)\rangle = \frac{1}{n}\sum_{i=0}^{n-1}\langle f(X)\rangle = \langle f(X)\rangle \tag{7.180}$$

In the first equality we substituted the definition of the sample mean from Eq. (7.179). In the second equality we used (repeatedly) the addition rule for expectations of random variables, from Appendix C.3. In the third equality we noticed that all n terms in the sum are identical and therefore cancelled the denominator. In words, our result is that *the expectation of the sample mean is equal to the population mean*. This motivates our choice to use $\overline{f}$ as an *estimator* of $\langle f(X)\rangle$; for us an estimator is a useful approximation of a given quantity. In a problem you will prove the *weak law of large numbers*, which tells us that as n gets larger there is a high probability that the sample mean will be close to the population mean. This is good news, but it doesn't tell us the full story: we would like to know how *fast* the sample mean approaches the population mean, so we can decide when to stop.

In order to quantify how well we're doing, we turn to the *variance of the sample mean*:

$$\mathrm{var}(\overline{f}) = \mathrm{var}\left(\frac{1}{n}\sum_{i=0}^{n-1} f(X_i)\right) = \frac{1}{n^2}\sum_{i=0}^{n-1}\mathrm{var}[f(X)] = \frac{1}{n}\mathrm{var}[f(X)] \tag{7.181}$$

In the first equality we substituted the definition of the sample mean from Eq. (7.179). In the second equality we used (repeatedly) the addition rule for variances of random variables, from Appendix C.3. In the third equality we noticed that all n terms in the sum are identical and therefore cancelled one of the n's in the denominator. Since $\mathrm{var}[f(X)]$ is fixed and given by Eq. (7.178), our result is telling us that *the variance of the sample mean decreases as* $1/n$. If we take the square root on both sides, we find:

$$\sqrt{\mathrm{var}(\overline{f})} = \frac{1}{\sqrt{n}}\sqrt{\mathrm{var}[f(X)]} \tag{7.182}$$

that is, the standard deviation of the sample mean goes as $1/\sqrt{n}$: as you quadruple n, the standard deviation is halved. As you will discover in a later section, this scaling persists in the multidimensional version of Monte Carlo; as we saw in our discussion following

[36] Since the sample mean is a random variable, we take its expectation to see if it gives the population mean.

Eq. (7.163), this is why Monte Carlo vastly outperforms conventional quadrature methods when integrating over many dimensions.

There is a slight complication, though: Eq. (7.181) relates $\text{var}(\overline{f})$ (the variance of the sample mean) to $\text{var}[f(X)]$ (the population variance) which, as you may recall from our discussion around Eq. (7.178), we don't actually know. This means that we will need to come up with an estimator for $\text{var}[f(X)]$. Once we have a usable approximation for it, we can employ Eq. (7.181) to produce an actual number that approximates the variance of the sample mean. Given the standard definition of the variance of a random variable, we propose an estimator for the population variance of the following form:

$$e_{var} = \overline{f^2} - \overline{f}^2 = \frac{1}{n}\sum_{i=0}^{n-1} f^2(X_i) - \left[\frac{1}{n}\sum_{i=0}^{n-1} f(X_i)\right]^2 \tag{7.183}$$

which introduces the new notation $\overline{f^2}$, whose meaning should be intuitive. It's important to note that the final expression makes use of the already observed values of $f(X_i)$; this means that it can readily be evaluated. As you will show in a problem, the expectation of this new estimator e_{var} is close to, but not quite the same as, the population variance:[37]

$$\langle e_{var}\rangle = \frac{n-1}{n}\text{var}[f(X)] \tag{7.184}$$

This implies that it is *not* an *unbiased estimator* (i.e., it is a biased estimator). An unbiased estimator g of a quantity G is one for which the mean is equal to the quantity you're trying to estimate, $\langle g\rangle = G$.[38] It's trivial to see how to produce an unbiased estimator of the variance: simply multiply both sides with n and divide with $n-1$, i.e., $ne_{var}/(n-1)$ is an unbiased estimator of the population variance (known as the *sample variance*). We can now use this unbiased estimator to get:

$$\text{var}(\overline{f}) = \frac{1}{n}\text{var}[f(X)] \approx \frac{1}{n}\frac{n}{n-1}e_{var} = \frac{1}{n-1}\left(\overline{f^2} - \overline{f}^2\right) \tag{7.185}$$

The first step is Eq. (7.181), namely the relationship between the variance of the sample mean, on the one hand, and the population variance, on the other. The second step employs the unbiased estimator of the population variance, i.e., the sample variance. The third step cancelled the n and plugged in e_{var} from Eq. (7.183). To summarize, we approximate the population variance $\text{var}[f(X)]$ by the sample variance, $ne_{var}/(n-1)$; then, we use that in Eq. (7.181) to produce an estimate for the variance of the sample mean.[39]

Practical Prescription

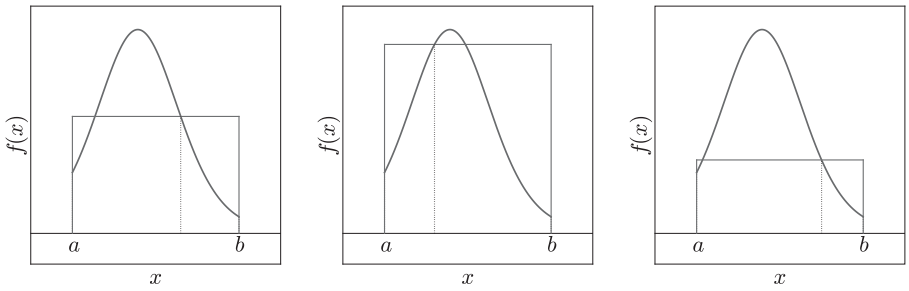
Let's back up for a second, to recall that our goal all this while has been to approximate an integral. We can trivially recast Eq. (7.176) as:

$$\int_a^b f(x)dx = (b-a)\langle f(X)\rangle \tag{7.186}$$

[37] Remember: e_{var} is an estimator, so we take its expectation to see if it gives the population variance.

[38] Obviously, for large values of n we have $(n-1)/n \approx 1$, so $\langle e_{var}\rangle$ is basically equal to $\text{var}[f(X)]$.

[39] You may wish to re-read the last few pages at this point, as the (standard) terminology can be somewhat confusing. For example, the "variance of the sample mean" is not the same thing as the "sample variance".



Randomly chosen abscissas, each of which leads to a rectangle **Fig. 7.6**

The previous subsection studied the approximation of the population mean $\langle f(X)\rangle$ by the sample mean $\overline{f}$, Eq. (7.179). In terms of our integral, this gives:

$$\int_a^b f(x)dx \approx (b-a)\overline{f} \pm (b-a)\sqrt{\text{var}(\overline{f})} \tag{7.187}$$

We took the square root of the variance of the sample mean here to produce the standard deviation of the sample mean; in order to interpret that in the usual (Gaussian) sense of $\pm$, one also needs to make the assumption of a finite variance, which will always be true for us. In other words, we employed the *central limit theorem* which tells us that asymptotically the $\overline{f}$ obeys a normal/Gaussian distribution regardless of which $p(x)$ was employed to draw the samples (e.g., a uniform one in our case); this is derived in a problem.

Using Eq. (7.185) for the variance of the sample mean, we find:

$$\int_a^b f(x)dx \approx (b-a)\overline{f} \pm (b-a)\sqrt{\frac{\overline{f^2}-\overline{f}^2}{n-1}} \tag{7.188}$$

If we also take the opportunity to plug in $\overline{f}$ and $\overline{f^2}$ from Eq. (7.179) and Eq. (7.183), respectively, we arrive at the following practical formula for *Monte Carlo integration*:

$$\int_a^b f(x)dx \approx \frac{b-a}{n}\sum_{i=0}^{n-1} f(X_i) \pm \frac{b-a}{\sqrt{n-1}}\sqrt{\frac{1}{n}\sum_{i=0}^{n-1} f^2(X_i) - \left[\frac{1}{n}\sum_{i=0}^{n-1} f(X_i)\right]^2} \tag{7.189}$$

where the X_i's are chosen uniformly from a to b.

We will implement Eq. (7.189) in Python later, where we will see that the $\sqrt{n-1}$ in the denominator has the effect of slowly but surely decreasing the standard deviation of the sample mean: more work pays off. For now, we limit ourselves to the first term on the right-hand side: observe that this is an arithmetic average over terms of the form $(b-a)f(X_i)$. This is similar to the rectangle or midpoint methods, Fig. 7.2, in that the total area is evaluated as a sum of rectangle areas. The difference here is that the width of the rectangles is always $b-a$ (i.e., each term corresponds to a big rectangle) and the height is determined by the value of the function at the uniformly distributed X_i's. This is illustrated in Fig. 7.6.

7.7.3 Monte Carlo beyond the Uniform Distribution

Having understood how and why Monte Carlo quadrature works, we now see if we can solve harder problems, or solve the same problems better. To give the punchline ahead of time, this will involve non-uniformly distributed random numbers, a subject to which we return in section 7.7.5 below, when we introduce the *Metropolis–Hastings algorithm*.

Generalizing to Weight Functions

The discussion in the previous section started from Eq. (7.175), which gives us the expectation of a function of a continuous random variable:

$$\langle f(X)\rangle = \int_{-\infty}^{+\infty} p(x)f(x)dx \tag{7.190}$$

This time around, we take our probability density function to be $w(x)/(b-a)$ from a to b and zero elsewhere; this $w(x)$ is kept general, except for being positive. This leads to:[40]

$$\langle f(X)\rangle = \frac{1}{b-a}\int_{a}^{b} w(x)f(x)dx \tag{7.191}$$

Notice that the integral on the right-hand side is precisely of the form we encountered in Eq. (7.129), in our discussion of general Gaussian quadrature. It should therefore come as no surprise that $w(x)$ is known as a *weight function*. For this new population mean, we can write down the corresponding population variance; this follows from Eq. (7.177) and would lead to a generalization of Eq. (7.178), where this time the integrands would be $[w(x)f(x)]^2$ and $w(x)f(x)$.

We could now introduce a new sample mean, as per Eq. (7.179):

$$\overline{f} = \frac{1}{n}\sum_{i=0}^{n-1} f(X_i) \tag{7.192}$$

where, crucially, this time the $X_0, X_1, X_2, \ldots, X_{n-1}$ are drawn randomly *from the probability density function* $w(x)$, i.e., *not* from a uniform distribution. Intriguingly, if you now go back to our derivation on the expectation of the sample mean, Eq. (7.180), you can easily see that all the steps carry over in exactly the same way. The only thing that's changed is the meaning of the expectation, which this time is given as per Eq. (7.191). Similarly, the derivation on the variance of the sample mean is also identical to Eq. (7.181) and, finally, the same holds for the expectation of our proposed estimator for the population variance, Eq. (7.184). As before, the only thing that's changed is the meaning of the expected value. To make a long story short, the entire argument carries over, implying that we have a practical formula for *Monte Carlo integration with a general weight function*:

[40] We write things this way in order to ensure that $w(x) = 1$ takes us back to Eq. (7.176). Note that this implies the following normalization of the weight function: $\int_{-\infty}^{+\infty} p(x)dx = 1$, so $\int_a^b w(x)dx = b-a$.

$$\int_a^b w(x)f(x)dx \approx \frac{b-a}{n}\sum_{i=0}^{n-1} f(X_i) \pm \frac{b-a}{\sqrt{n-1}}\sqrt{\frac{1}{n}\sum_{i=0}^{n-1} f^2(X_i) - \left[\frac{1}{n}\sum_{i=0}^{n-1} f(X_i)\right]^2} \quad (7.193)$$

where this time the X_i's are chosen from $w(x)$, e.g., if $w(x)$ is exponential, these are exponentially distributed random numbers.

Inverse Transform Sampling

In the preceding discussion we took it for granted that the random variables $X_0, X_1, X_2, \ldots, X_{n-1}$ could be drawn from the probability density function $w(x)$. You may have wondered how that is possible, given that section 7.7.1 only discussed the generation of uniformly distributed random numbers. We now go over a specific technique that helps you accomplish this task, known as *inverse transform sampling* or, sometimes, simply *inverse sampling*.

The main idea is to somehow manage to convert the integrand wf in Eq. (7.193) to f, via an appropriate change of variables; in calculus, this is known as *integration by substitution*. As advertised, we wish to somehow "absorb" the $w(x)$; with that in mind, we make the following change of variables:[41]

$$du = w(x)dx \quad (7.194)$$

This can be integrated:

$$u(x) = \int_a^x w(x')dx' \quad (7.195)$$

If you have taken a course on probability, you will recognize this integral over the probability density function as the *cumulative distribution function*.[42]

We now realize that when x goes from a to b, u goes from 0 to $b-a$, i.e., $u(a) = 0$ and $u(b) = b-a$. Since Eq. (7.195) gives us u as a function of x, i.e., $u(x)$, we can (hope to be able to) invert this to give x as a function of u, i.e., $x(u)$. Having done that, we realize that we've been able to recast the integral over x as an integral over u:

$$\int_a^b w(x)f(x)dx = \int_0^{b-a} f\left(x(u)\right)du \quad (7.196)$$

where $f(x(u))$ means that we've expressed x in terms of u. Roughly speaking, in Eq. (7.194) we equated the differentials, so we needed to integrate that relation in order to change $f(x)$ into something that depends on u.

At this point, we realize that the right-hand side is in the form of an *unweighted* integral

[41] You can think of Eq. (7.194) as equating two probabilities, $p_u(u)du = p_x(x)dx$, where $p_u(u)$ is 1.

[42] In probability theory, this would be $\int_{-\infty}^{x} p(x)dx$, but our $p(x)$ is 0 below a; notice that we are using $w(x)$, so our normalization is different (because of the $b-a$ term).

over u, implying that we can use Eq. (7.189) to evaluate it by employing U_i's that are chosen uniformly:

$$\int_a^b w(x)f(x)dx = \int_0^{b-a} f\left(x(u)\right)du \approx \frac{b-a}{n}\sum_{i=0}^{n-1} f\left(X(U_i)\right) \tag{7.197}$$

To reiterate, the U_i's are uniformly distributed from 0 to $b-a$ and as a result the X_i's (which are produced by our having inverted $u(x)$ to give $x(u)$) are distributed according to $w(x)$, from a to b; basically, the X_i's are $w(x)$-aware, via the inverse transform we carried out. In summary, we've managed to employ uniform numbers, together with a transformation, to apply Monte Carlo to a weighted integral.

To see how this all works in practice, let's study a specific example:

$$I = \int_0^1 e^{-x}\cos x dx \tag{7.198}$$

We could choose to treat the entire integrand as one piece here, $f(x) = e^{-x}\cos x$; we could then directly apply Eq. (7.189) and use uniformly distributed random numbers to compute it. While this is certainly an option, you can see why it is a wasteful one: the exponential causes a considerable drop in the magnitude of the integrand; by choosing your abscissas uniformly, you are not taking into consideration the fact that most of the contribution to the integral comes from small values of x.

Thus, another choice is to try to apply Eq. (7.193) instead of Eq. (7.189); since you want to ensure your weight function is normalized, you take $w(x) = ce^{-x}$, where the normalization gives $c = e/(e-1)$. To get the prefactors to cancel, you should take $f(x) = \cos x/c$. The problem you then face is that you don't have access to a random-number generator for exponentially distributed numbers.[43] The inverse transform method then tells you to evaluate $u(x)$ as per Eq. (7.195):

$$u(x) = c\int_0^x e^{-x'}dx' = c(1-e^{-x}) \tag{7.199}$$

This can be inverted to give:

$$x(u) = -\ln\left(1-\frac{u}{c}\right) \tag{7.200}$$

in which case Eq. (7.197) for the integral in Eq. (7.198) becomes:

$$\int_0^1 ce^{-x}\frac{1}{c}\cos x dx = \int_0^1 \frac{1}{c}\cos\left(-\ln\left[1-\frac{u}{c}\right]\right)du \approx \frac{1}{n}\sum_{i=0}^{n-1}\frac{1}{c}\cos\left(-\ln\left[1-\frac{U_i}{c}\right]\right) \tag{7.201}$$

where the U_i's are distributed uniformly from 0 to 1; a problem asks you to implement this in Python. In short, you have managed to sample from an exponential distribution, even though your input random-number generator was for a uniform distribution.

[43] Actually, you have access to `numpy.random.exponential()`, but let's pretend you don't.

Importance Sampling

You may be thinking that it was easy to handle an integrand of the form $e^{-x}\cos x$, with its clean separation between an exponential suppression and everything else. This was basically the integrand itself telling us which weight function to employ. As it turns out, the technique we just introduced is much more useful than we've hinted at so far: even if no weight function is present, you can introduce one yourself.

To see what we mean, let's repeat the previous argument, for the case where the integrand appears to be a single (i.e., unweighted) function:

$$\int_a^b f(x)dx = \int_a^b w(x)\frac{f(x)}{w(x)}dx = \int_0^{b-a}\frac{f\left(x(u)\right)}{w\left(x(u)\right)}du \approx \frac{b-a}{n}\sum_{i=0}^{n-1}\frac{f\left(X(U_i)\right)}{w\left(X(U_i)\right)} \tag{7.202}$$

In the first equality we multiplied and divided with a positive weight function of our choosing. In the second equality we used the change of variables from Eq. (7.195), in complete analogy to what we saw in Eq. (7.196); the only difference is that this time after we carry out the transform there is still a w left over in the denominator. In the third equality we treated f/w as our ("unweighted") integrand and therefore used U_i's which are uniformly distributed from 0 to $b-a$. The entire process is known as *importance sampling*.

You may be wondering why we went through the trouble of doing this. The answer is as follows: if you choose a $w(x)$ that behaves approximately the same way that $f(x)$ does, then your random numbers will be distributed in the most "important" regions, instead of uniformly. To put the same idea differently: since our (unweighted) integrand is f/w, the variance will be computed as per Eq. (7.189), with f/w in the place of f. Since $w(x)$ is chosen to be similar to $f(x)$, we see that f/w will be less varying than f itself was; this will lead to a reduction of the variance, i.e., a better overall estimate of the integral we are trying to compute.

To see this in action, we turn to the example we've been revisiting throughout this chapter, namely the electrostatic potential in Eq. (7.2), which integrates the function:

$$f(x) = \frac{1}{\sqrt{x^2+1}} \tag{7.203}$$

from 0 to 1. As advertised, this integral doesn't consist of two easily separable parts. Even so, we can plot it to see what it looks like; this is done in the left panel of Fig. 7.7. We see that $f(x)$ is decreasing from 1 to 0.7 in our interval; with that in mind, we decide to employ a linear weight function:

$$w(x) = c_0 + c_1 x \tag{7.204}$$

We wish $w(x)$ to roughly track the behavior of $f(x)$ in our interval; one way to do this is to ensure that $f(0)/w(0)$ is equal to $f(1)/w(1)$. This gives one equation relating c_0 and c_1. If we then also impose the normalization condition $\int_0^1 w(x)dx = 1$ we get another relation. Thus, we are able to determine both parameters:

$$c_0 = 4 - 2\sqrt{2}, \qquad c_1 = -6 + 4\sqrt{2} \tag{7.205}$$

It comes as no surprise that c_1 is negative, since $w(x)$ should be a decreasing function in our interval. Our weight function is also plotted in the left panel of Fig. 7.7, together with the ratio f/w: we see that $f(x)/w(x)$ varies between 0.85 and 0.9, which is considerably smaller than the variation of our original integrand. It will therefore not come as a surprise in the following subsection that the computed variance in the importance-sampling case turns out to be much smaller than the variance in the original, uniform-sampling case.

Backing up for a second, what we're interested in doing is applying the importance-sampling prescription in Eq. (7.202). Up to this point, we've only selected the $w(x)$ we'll use. The next step is to carry out the change of variables as per Eq. (7.195):

$$u(x) = \int_0^x (c_0 + c_1 x')dx' = c_0 x + \frac{1}{2} c_1 x^2 \tag{7.206}$$

After that, we need to invert this relation to find $x(u)$; since we're dealing with a quadratic, we will have two roots. Of these, we pick the one that leads to x from 0 to 1 for u in $[0, 1]$, since that's the interval we're interested in. We get:

$$x(u) = \frac{-c_0 + \sqrt{2c_1 u + c_0^2}}{c_1} \tag{7.207}$$

Having calculated both the weight $w(x)$ and the inverted $x(u)$, we are now ready to apply importance sampling to our problem, as per Eq. (7.202):

$$\int_0^1 \frac{1}{\sqrt{x^2+1}} dx \approx \frac{1}{n} \sum_{i=0}^{n-1} \frac{1}{\sqrt{X(U_i)^2+1}\,(c_0 + c_1 X(U_i))} \tag{7.208}$$

where we get the $X(U_i)$ from Eq. (7.207). As before, the U_i's are uniformly distributed. This is implemented in Python in the following section, where we will discover if all our effort was worthwhile.

Incidentally, you may be wondering why we picked a linear $w(x)$ in Eq. (7.204). As the left panel of Fig. 7.7 clearly shows, our integrand itself is not linear. Thus, you may feel tempted to pick a $w(x)$ that is slightly more complicated (e.g., a quadratic), since this would imply that the variation in $f(x)/w(x)$ is even smaller. This is fine, in principle, but in practice you have to remember that the integral of $w(x)$, which you need to carry out for the change of variables as per Eq. (7.195), must always remain invertible; for example, if your $w(x)$ is quadratic then you will need to invert a cubic. As this example shows, as you introduce further structure into your $w(x)$, the inversion becomes increasingly hard to do analytically; of course, there's nothing holding you from trying to carry out this inversion numerically, but you will have to be careful about picking the right solution, as we saw in Eq. (7.207). To provide a quick preview of coming attractions, we note that this approach does not work that well if you try to generalize it to a multidimensional problem; we will discover a better alternative in section 7.7.5 below.

montecarlo.py Code 7.5

```
from newtoncotes import f
import numpy as np

def montecarlo(f,a,b,n,option="uniform"):
     np.random.seed(314159)
     us = np.random.uniform(a, b, n)

     if option=="uniform":
          fs = f(us)
     else:
          c0 = 4 - 2*np.sqrt(2)
          c1 = -6 + 4*np.sqrt(2)
          xs = (-c0 + np.sqrt(2*c1*us + c0**2))/c1
          fs = f(xs)/(c0 + c1*xs)

     fbar, err = stats(fs)
     return (b-a)*fbar, (b-a)*err

def stats(fs):
     n = fs.size
     fbar = np.sum(fs)/n
     fsq = np.sum(fs**2)/n
     varfbar = (fsq - fbar**2)/(n - 1)
     return fbar, np.sqrt(varfbar)

if __name__ == '__main__':
     for n in 10**np.arange(2,7):
          avu, erru = montecarlo(f, 0., 1., n)
          avi, erri = montecarlo(f, 0., 1., n, option="is")
          rowf = "{0:7d}  {1:1.9f} {2:1.9f}  {3:1.9f} {4:1.9f}"
          print(rowf.format(n, avu, erru, avi, erri))
```

7.7.4 Implementation

Code 7.5 implements Monte Carlo quadrature for the case of a one-dimensional integral. This is done in `montecarlo()`, which addresses both the case of uniform sampling and that of importance sampling. Crucially, uniform sampling is the default parameter value, which means that you can call this function with `montecarlo(f,a,b,n)`, i.e., with pre-

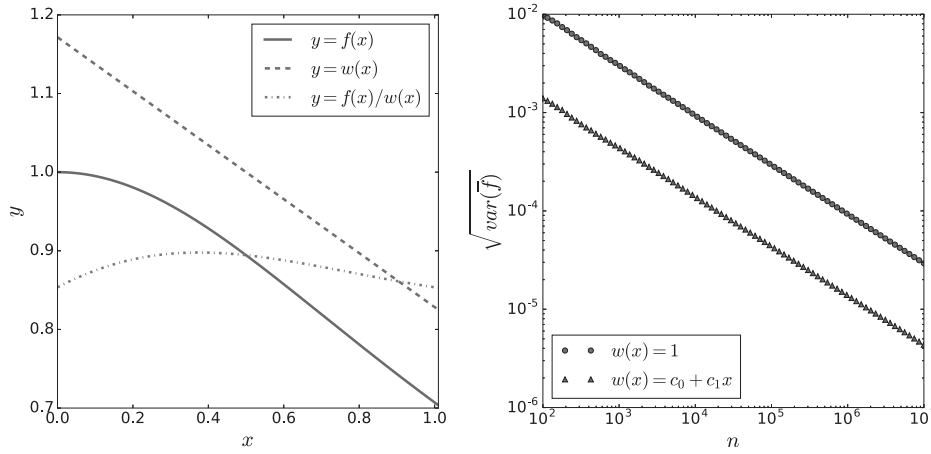


Fig. 7.7 The integrand, the weight, and their ratio (left); results of MC integration (right)

cisely the same interface as our three integrators in `newtoncotes.py` and the one in `gauleg.py`. In other words, this is a drop-in replacement for a general-purpose integration routine. In the function body, we start by seeding the random-number generator and producing an array of n numbers, uniformly distributed from a to b, using `numpy` functionality described in section 7.7.1. With these in hand, the option of uniform sampling is a straightforward application of Eq. (7.189). In the interest of separating our concerns, we have farmed out the evaluation of $\overline{f}$ and $\overline{f^2}$, which we need as per Eq. (7.188), to a separate function, `stats()`.[44]

For the importance-sampling case, we code up the parameters of the linear weight function, Eq. (7.205), then produce an array of X_i's using $X(U_i)$ from Eq. (7.207), and then we remember to divide $f(X(U_i))$ with $w(X(U_i))$, just like in Eq. (7.202) or Eq. (7.208). As always, we employ `numpy` functionality to carry out the division f/w for all i's in one line; that's why, as usual, the code contains no indices. The same holds, of course, with respect to `stats()`, where the squaring and the sums are carried out without needing explicit indices. Incidentally, Eq. (7.202) shows only the sample mean, not its variance; that is taken from our general formula for Monte Carlo integration, Eq. (7.189), with f/w in the place of f. Finally, we note that the uniformly distributed numbers for this case should in general go from 0 to $b - a$, not a to b; thus, our importance sampling option only works for $a = 0$ and $b = 1$. That's fine, though, since the $w(x)$ itself (i.e., both its values at the two endpoints and the normalization) was tuned only to the interval $[0, 1]$.

The main program picks n to be the first several powers of 10 and carries out Monte Carlo integration using either uniform sampling, Eq. (7.189), or importance sampling, Eq. (7.202), with a linear weight function. The output table format is then set up. For comparison, recall that the exact answer for this integral is `0.88137358702`. Running this code produces the following output:

[44] We could have used `numpy.mean()` and `numpy.std()` but, as usual, we prefer to roll our own.

```
   100    0.873135430  0.009827018    0.880184046  0.001397861
  1000    0.878313494  0.003014040    0.880653976  0.000439206
 10000    0.879343920  0.000933506    0.881029489  0.000139055
100000    0.881289768  0.000292906    0.881400087  0.000043577
1000000   0.881433836  0.000092589    0.881389786  0.000013775
```

First, looking at the uniform-sampling results we observe that they do converge to the right answer as n is increased; the exact answer is always within a few standard deviations of the sample mean (though often it lies within less than one standard deviation). However, it's fair to say that the convergence is quite slow: using a million points, the standard deviation is roughly 10^{-4}. As you may recall, Gauss–Legendre quadrature got all the printed digits right with only 10 points! Of course, this is an unfair comparison, since the true strength of Monte Carlo is in multidimensional integration, where even Gauss–Legendre can't help you, as we saw in Eq. (7.163).

We now turn to the importance-sampled results. Overall, we see that the importance sampling has helped reduce the standard deviation by roughly an order of magnitude, which was certainly worthwhile. Note that, in both cases, we are not showing the absolute error (i.e., comparing with the exact answer) but the standard deviation that comes out of the Monte Carlo machinery. Incidentally, the n increases by a factor of 10 as you move to the next row: as a result, we see the standard deviation get reduced by roughly a factor of 3 each time; this is consistent with a scaling of $1/\sqrt{n}$. In order to drive this point home, the right panel of Fig. 7.7 is a log-log plot of the standard deviation of the sample mean for the two cases of uniform sampling and importance sampling; we've used more n points to make the plot look good and that it does: the slope of -0.5 is pretty unmistakable in both cases. Just like we found in our table of values, we again see that importance sampling consistently helps us reduce the standard deviation by an order of magnitude.

7.7.5 Monte Carlo in Many Dimensions

We now turn to a truly relevant application of the Monte Carlo approach: multidimensional integration. As in the one-dimensional case, we start from uniform sampling, but then try to do a better job: we explain how to carry out weighted sampling via the *Metropolis–Hastings algorithm*, which is one of the most successful methods ever.

Uniform Sampling

We briefly talked about multidimensional integration in section 7.6.4, where we saw an example in Eq. (7.157):

$$I = \int_{-1}^{1} \int_{-1}^{1} f(x, y) dx dy \tag{7.209}$$

This notation can get cumbersome if we need many dimensions, so it makes more sense to employ notation like the one used in section 5.5.2; thus we bundle together the variables

$x_0, x_1, \ldots, x_{d-1}$ into $\mathbf{x}$. We will be dealing with a scalar function of many variables, i.e., $f(\mathbf{x})$ produces a single number when given d variables as input.[45] We emphasize that the number of dimensions (and therefore variables) is d. Our starting point is the same as in section 7.7.2, i.e., we define a *population mean*:

$$\langle f(\mathbf{X}) \rangle = \frac{1}{V} \int f(\mathbf{x}) d^d x \tag{7.210}$$

where the expectation on the left-hand side is of a function of a multidimensional random variable $\mathbf{X}$. The multidimensional integration on the right-hand side is over the volume V: this is the multidimensional generalization of writing $p(x) = 1/(b-a)$ in Eq. (7.176).

At this point, there is nothing discrete going on: we simply defined the population mean as the multidimensional integral in Eq. (7.210). We can now step through all the arguments in section 7.7.2, defining the population variance, the sample mean and its variance, testing that the sample mean is a good estimator of the population mean, and so on. The entire argument carries over naturally, leading to the following practical formula for *multidimensional Monte Carlo integration*:

$$\int f(\mathbf{x}) d^d x \approx \frac{V}{\mathcal{N}} \sum_{i=0}^{\mathcal{N}-1} f(\mathbf{X}_i) \pm \frac{V}{\sqrt{\mathcal{N}-1}} \sqrt{\frac{1}{\mathcal{N}} \sum_{i=0}^{\mathcal{N}-1} f^2(\mathbf{X}_i) - \left[\frac{1}{\mathcal{N}} \sum_{i=0}^{\mathcal{N}-1} f(\mathbf{X}_i) \right]^2} \tag{7.211}$$

which is a d-dimensional generalization of Eq. (7.189). Thus, the $\mathbf{X}_i$'s are chosen uniformly; this is a good time to unpack our notation a little. Each of the $\mathbf{X}_i$'s on the right-hand side is a d-dimensional random sample, whose components are $(\mathbf{X}_i)_0, (\mathbf{X}_i)_1, \ldots, (\mathbf{X}_i)_{d-1}$. In order to keep our notation consistent with that in section 7.6.4, in Eq. (7.211) we use $\mathcal{N}$ for the total number of samples; this plays the role n played in the one-dimensional sections.

As before, we've approximated the expectation of a continuous random variable (i.e., the population mean) with the arithmetic average of many function values taken at (discrete) uniformly distributed points. Each $\mathbf{X}_i$ is made up of d uniformly distributed random numbers; crucially, the values of a given $\mathbf{X}_i$ are totally unrelated to those of another sample (i.e., say, $\mathbf{X}_5$ is chosen independently of $\mathbf{X}_4$). It should therefore be straightforward to see how to implement this approach programmatically. A problem asks you to apply Eq. (7.211) to a 5-dimensional integral and compare with the corresponding five-dimensional Gauss–Legendre computation. The most significant feature of Eq. (7.211) is that the standard deviation of the sample mean goes as $1/\sqrt{\mathcal{N}-1}$, which is the standard Monte Carlo result that applies regardless of dimensionality: it involves $\mathcal{N}$ (the number of samples), not d (the dimensionality of each sample). This is why Monte Carlo is to be preferred over conventional quadrature for many-dimensional problems.

[45] So our f here plays the role of ϕ in section 5.5.2. Of course, now we're integrating, not minimizing.

Weighted Sampling via the Metropolis–Hastings Algorithm

As you may recall from section 7.7.3, uniform sampling is not the best we can do. While the simple prescription in Eq. (7.211) does work and does have the pleasant $1/\sqrt{\mathcal{N}-1}$ scaling, it is far from perfect: since the $\mathbf{X}_i$'s are chosen uniformly this may (and in practice does) imply that many random samples are wasted trying to evaluate $f(\mathbf{X}_i)$ in regions of the d-dimensional space where not much is going on. It would be nice to have a Monte Carlo technique where the random samples, i.e., the $\mathbf{X}_i$'s, are preferentially chosen to be such that they can make a difference: an integrand-aware sampling would be considerably more efficient than uniformly producing random samples.

You may have noticed a pattern in our discussion of Monte Carlo techniques: the main feature (approximate the population mean via the sample mean) is always the same. Thus, we can straightforwardly generalize the argument in section 7.7.3 to produce the following practical formula for *multidimensional weighted Monte Carlo integration*:

$$\int w(\mathbf{x}) f(\mathbf{x}) d^d x \approx \frac{V}{\mathcal{N}} \sum_{i=0}^{\mathcal{N}-1} f(\mathbf{X}_i) \pm \frac{V}{\sqrt{\mathcal{N}-1}} \sqrt{\frac{1}{\mathcal{N}} \sum_{i=0}^{\mathcal{N}-1} f^2(\mathbf{X}_i) - \left[\frac{1}{\mathcal{N}} \sum_{i=0}^{\mathcal{N}-1} f(\mathbf{X}_i)\right]^2} \quad (7.212)$$

where this time the $\mathbf{X}_i$'s are drawn from $w(\mathbf{x})$, e.g., if $w(\mathbf{x})$ is exponential, these are samples made up of exponentially distributed random numbers. Note that, just like $f(\mathbf{x})$, the weight function $w(\mathbf{x})$ is now a function of d variables. If you're interested in integrating over all of space, simply drop the V term; to see why, picture Eq. (7.191) for the case of integrating from $-\infty$ to $+\infty$: as per Eq. (7.190), you would have $p(x) = w(x)$ and no $(b-a)$ term would appear on the right-hand side.

For the one-dimensional case, when we were faced with Eq. (7.193), we employed the inverse transform sampling method to produce X_i's which were drawn from the probability density function $w(x)$. In other words, we made the change of variables in Eq. (7.194), integrated as per Eq. (7.195), then inverted, and were able to go from uniform samples to weighted samples, Eq. (7.197). When you move on to a multidimensional problem, i.e., producing random samples drawn from $w(\mathbf{x})$, our trivial change of variables would have to be replaced by its generalization, namely the determinant of the Jacobian matrix. To make matters worse, the generalization of the integral in Eq. (7.195) would have to be carried out somehow (analytically). As if that weren't enough, the inversion (in, say, 100 variables) cannot be carried out analytically; thus, you would have to employ one of the multidimensional root-finders from chapter 5. As you know by now, these approaches are prone to fail if you don't start sufficiently near the root. In a multidimensional problem, you will also likely have many roots and would be faced with the task of automatically selecting the correct one. Putting it all together, this approach is not practical when the number of dimensions is large. While other techniques exist to carry out weighted sampling in one-dimensional problems, they also get in trouble when the dimensionality increases.

Markov Chains

Basically the only thing that works as far as multidimensional weighted sampling is concerned is *Markov chain Monte Carlo* (MCMC).[46] The main new concept involved here is that of a *Markov chain*: imagine you have a sequence of random samples $\mathbf{X}_0, \mathbf{X}_1, \ldots, \mathbf{X}_{\mathcal{N}-1}$ for which a given sample $\mathbf{X}_i$ depends only on the previous one, i.e., on $\mathbf{X}_{i-1}$, but not on any of the earlier ones, i.e., $\mathbf{X}_{i-2}$, $\mathbf{X}_{i-3}$, and so on. In other words, one starts from a random sample $\mathbf{X}_0$, uses that to produce sample $\mathbf{X}_1$, then uses that in its turn to produce sample $\mathbf{X}_2$, and so on, always using the current sample to create the next one. This sequence of samples, $\mathbf{X}_0, \mathbf{X}_1, \ldots, \mathbf{X}_{\mathcal{N}-1}$, is known as a *random walk*.[47] Note that this is quite different from what we were doing in earlier sections: there the X_i (or the U_i) were independent from one another, whereas now we use a given $\mathbf{X}_{i-1}$ to produce the next one, i.e., $\mathbf{X}_i$. The reason Markov chains are so useful is that they can be produced such that they asymptotically (i.e., as $\mathcal{N} \to \infty$) have the distribution we would like them to, which in our case would be $w(\mathbf{x})$. One could therefore do an increasingly better job at computing a d-dimensional integral by continuing the Markov chain for larger values of $\mathcal{N}$.

Detailed Balance

We wish to produce a Markov chain with an asymptotic distribution of our choosing, which would therefore be the stationary distribution of the chain (if you don't know what that means, keep reading). Thus, we can borrow ideas from the statistical mechanics of systems in equilibrium. A sufficient (but not necessary) condition of evolving toward equilibrium and staying there is the *principle of detailed balance*:

$$w(\mathbf{X})T(\mathbf{X} \to \mathbf{Y}) = w(\mathbf{Y})T(\mathbf{Y} \to \mathbf{X}) \tag{7.213}$$

Here $T(\mathbf{X} \to \mathbf{Y})$ is the (conditional) probability density that you will move to $\mathbf{Y}$ if you start at $\mathbf{X}$; it is often called the *transition probability*.[48] Since we're dealing with a Markov chain, we need to know how to go from one sample to the next; this is precisely what the transition probability will allow us to do, soon. Since $w(\mathbf{X})$ is the probability density of being near $\mathbf{X}$, $w(\mathbf{X})T(\mathbf{X} \to \mathbf{Y})$ quantifies how likely it is overall to start at $\mathbf{X}$ and move to $\mathbf{Y}$. Similarly, $w(\mathbf{Y})T(\mathbf{Y} \to \mathbf{X})$ tells us how likely it is to start at $\mathbf{Y}$ and move to $\mathbf{X}$. In words, Eq. (7.213) says that it is equally likely that we will go in one direction as in the reverse direction. The principle of detailed balance is sometimes known as the *reversibility condition*, due to the fact that the reverse process would result if everything went backward in time. Intuitively, detailed balance tells us that if you're in equilibrium then effectively not much is changing: you could go somewhere, but you're just as likely to come back.

[46] For a rigorous treatment, have a look at the bibliography. Keep in mind that different authors mean different things when they use terms like "recurrent", "ergodic", and so on.

[47] In practice, one employs an ensemble of Markov chains, starting from several independent initial configurations; this gives rise to distinct *walkers*, but we'll stick to a single Markov chain for the sake of simplicity.

[48] More generally, one often says "probability" instead of "probability density"; this is analogous to classical field theory, where many times people will say "Lagrangian" even when they actually mean "Lagrangian density".

At this stage, detailed balance is just a condition: we haven't shown how to actually produce a Markov chain that obeys it. Even so, we will now spend some time seeing exactly how the detailed-balance condition can help us accomplish our goal. Instead of thinking about moving from an individual sample to another one, it can be helpful to think in terms of going from one probability density function to another. Assume that $p_{i-1}(\mathbf{X})$ is the distribution of values of the random variable $\mathbf{X}_{i-1}$ and, similarly, $p_i(\mathbf{X})$ is the distribution of $\mathbf{X}_i$. We can straightforwardly relate $p_i(\mathbf{X})$ to $p_{i-1}(\mathbf{X})$ as follows:[49]

$$p_i(\mathbf{X}) = p_{i-1}(\mathbf{X}) + \int \left[p_{i-1}(\mathbf{Y})T(\mathbf{Y} \to \mathbf{X}) - p_{i-1}(\mathbf{X})T(\mathbf{X} \to \mathbf{Y})\right] d^d Y \tag{7.214}$$

In words, what this is saying is that the probability of being near $\mathbf{X}$ at step i is equal to the probability of being near $\mathbf{X}$ at step $i-1$, plus the probability of leaving all other configurations $\mathbf{Y}$ and coming to $\mathbf{X}$, minus the probability of leaving $\mathbf{X}$ and going to any other configurations $\mathbf{Y}$. We will now put Eq. (7.214) together with the condition of detailed balance, Eq. (7.213), to see what we get.

First, we will show that if Eq. (7.213) holds, then Eq. (7.214) shows that $w(\mathbf{X})$ is a fixed point of the iteration (see section 5.2.3). For $p_{i-1}(\mathbf{X}) = w(\mathbf{X})$, Eq. (7.214) turns into:

$$p_i(\mathbf{X}) = w(\mathbf{X}) + \int \left[w(\mathbf{Y})T(\mathbf{Y} \to \mathbf{X}) - w(\mathbf{X})T(\mathbf{X} \to \mathbf{Y})\right] d^d Y = w(\mathbf{X}) \tag{7.215}$$

In the last step we noticed that the integrand vanishes, due to the detailed balance condition. Thus, we have shown that $p_i(\mathbf{X}) = p_{i-1}(\mathbf{X}) = w(\mathbf{X})$, i.e., we are dealing with a stationary distribution. This is saying that if at some point of the iteration our distribution becomes the desired weight distribution, then it will stay there.

Second, we would like to know that we are actually approaching that stationary distribution: it wouldn't do us much good if a fixed point existed but we could never reach it.[50] To see this, divide Eq. (7.214) by the desired distribution, $w(\mathbf{X})$:[51]

$$\begin{aligned}\frac{p_i(\mathbf{X})}{w(\mathbf{X})} &= \frac{p_{i-1}(\mathbf{X})}{w(\mathbf{X})} + \int \left[p_{i-1}(\mathbf{Y})\frac{T(\mathbf{Y} \to \mathbf{X})}{w(\mathbf{X})} - p_{i-1}(\mathbf{X})\frac{T(\mathbf{X} \to \mathbf{Y})}{w(\mathbf{X})}\right] d^d Y \\ &= \frac{p_{i-1}(\mathbf{X})}{w(\mathbf{X})} + \int T(\mathbf{X} \to \mathbf{Y}) \left[\frac{p_{i-1}(\mathbf{Y})}{w(\mathbf{Y})} - \frac{p_{i-1}(\mathbf{X})}{w(\mathbf{X})}\right] d^d Y \end{aligned} \tag{7.216}$$

To get to the second line, we first used the detailed balance condition, Eq. (7.213), to replace $T(\mathbf{Y} \to \mathbf{X})/w(\mathbf{X})$ by $T(\mathbf{X} \to \mathbf{Y})/w(\mathbf{Y})$ and then pulled out the common term $T(\mathbf{X} \to \mathbf{Y})$. Observe that the two terms inside the square brackets are expressed in terms of "actual distribution" divided by "desired distribution", i.e., p_{i-1}/w. Keep in mind that $T(\mathbf{X} \to \mathbf{Y})$ is a conditional probability density and is therefore non-negative. Think about what Eq. (7.216) tells us will happen to a $p_{i-1}(\mathbf{X})/w(\mathbf{X})$ that is near a maximum, i.e., is larger than other ratios, $p_{i-1}(\mathbf{Y})/w(\mathbf{Y})$: the square bracket will be negative, therefore the

[49] Looking at the equivalent relation Eq. (7.257) may help you solidify your intuition.

[50] We also need to (and will) satisfy a further condition at this point: it should be possible for the random walk to return to the neighborhood of a given $\mathbf{X}$, but this should not happen periodically.

[51] The argument can be modified to employ differences between these entities (instead of ratios).

entire right-hand side will drive this ratio down. Correspondingly, if $p_{i-1}(\mathbf{X})/w(\mathbf{X})$ is near a minimum, i.e., is smaller than the other ratios, the square bracket will be positive, therefore the entire right-hand side will drive this ratio up. In both cases, the ratio $p_i(\mathbf{X})/w(\mathbf{X})$ will be closer to 1 than $p_{i-1}(\mathbf{X})/w(\mathbf{X})$ was.

While we *still* haven't shown how to produce a Markov chain that obeys detailed balance, our two results in Eq. (7.215) and Eq. (7.216) are that *if* you have a Markov chain that obeys detailed balance then: (a) $w(\mathbf{X})$ *is a stationary distribution*, and (b) $p_i(\mathbf{X})$ *asymptotically approaches that stationary distribution*. In other words, our Markov chain will approach a d-dimensional equilibrium distribution of our choosing. We will now introduce an elegant trick that is able to produce a Markov chain obeying detailed balance.

Metropolis–Hastings Algorithm

The *Metropolis–Hastings algorithm* starts by splitting the transition probability:

$$T(\mathbf{X} \to \mathbf{Y}) = \pi(\mathbf{X} \to \mathbf{Y})\alpha(\mathbf{X} \to \mathbf{Y}) \tag{7.217}$$

where $\pi(\mathbf{X} \to \mathbf{Y})$ is the probability of making a *proposed* step from $\mathbf{X}$ to $\mathbf{Y}$ and $\alpha(\mathbf{X} \to \mathbf{Y})$ is the probability of *accepting* that move. Note that since we are dealing with an *acceptance probability* $\alpha(\mathbf{X} \to \mathbf{Y})$, this means that some moves will be accepted (i.e., the system moves from $\mathbf{X}$ to $\mathbf{Y}$) and some moves will be rejected (i.e., the system will stay at $\mathbf{X}$). The *proposal probability* $\pi(\mathbf{X} \to \mathbf{Y})$ is not unique, and several choices are discussed in the literature. The acceptance probability will be chosen in such a way that detailed balance is obeyed.

The Metropolis–Hastings algorithm proceeds by evaluating the following quantity:

$$R(\mathbf{X} \to \mathbf{Y}) = \frac{w(\mathbf{Y})}{w(\mathbf{X})}\frac{\pi(\mathbf{Y} \to \mathbf{X})}{\pi(\mathbf{X} \to \mathbf{Y})} \tag{7.218}$$

which is sometimes known as the *Metropolis–Hastings ratio*. Note that everything on the right-hand side is known: the desired distribution w is of our choosing, as is also true of the proposal distribution π. As a matter of fact, a simpler version of the Metropolis–Hastings algorithm, known as the *Metropolis algorithm* since that's how it was originally put forward, employs a symmetric proposal distribution; when $\pi(\mathbf{X} \to \mathbf{Y}) = \pi(\mathbf{Y} \to \mathbf{X})$ you can see that the ratio is simply $R(\mathbf{X} \to \mathbf{Y}) = w(\mathbf{Y})/w(\mathbf{X})$, namely the ratio of the (analytically known) desired weight at the configuration $\mathbf{Y}$ and at the configuration $\mathbf{X}$.

The next part of the Metropolis–Hastings algorithm is to use the ratio $R(\mathbf{X} \to \mathbf{Y})$ to determine the acceptance probability as follows:

$$\alpha(\mathbf{X} \to \mathbf{Y}) = \min\left[1, R(\mathbf{X} \to \mathbf{Y})\right] \tag{7.219}$$

We already know that $R(\mathbf{X} \to \mathbf{Y})$ is non-negative. What Eq. (7.219) does is to account for the possibility that the ratio $R(\mathbf{X} \to \mathbf{Y})$ is larger than 1: in that case, the acceptance probability is taken to be 1 (i.e., the step is guaranteed to be taken). If $R(\mathbf{X} \to \mathbf{Y})$ is less than 1, then the proposed step is taken with probability $\alpha(\mathbf{X} \to \mathbf{Y})$ (i.e., it is not taken with probability $1 - \alpha(\mathbf{X} \to \mathbf{Y})$).

As Eq. (7.219) now stands, it may not be totally obvious why we said that the acceptance probability was selected so that we can satisfy detailed balance. It's true that the evaluation of α involves only π and w, both of which are known, but what does this have to do with detailed balance? We will now explicitly answer this question; as you will see, showing this is quite easy, now that you are comfortable with all the concepts involved. Let us start with the left-hand side of the detailed-balance condition, Eq. (7.213):

$$\begin{aligned} w(\mathbf{X})T(\mathbf{X} \to \mathbf{Y}) &= w(\mathbf{X})\pi(\mathbf{X} \to \mathbf{Y})\alpha(\mathbf{X} \to \mathbf{Y}) = w(\mathbf{X})\pi(\mathbf{X} \to \mathbf{Y})\frac{w(\mathbf{Y})}{w(\mathbf{X})}\frac{\pi(\mathbf{Y} \to \mathbf{X})}{\pi(\mathbf{X} \to \mathbf{Y})} \\ &= w(\mathbf{Y})\pi(\mathbf{Y} \to \mathbf{X})\alpha(\mathbf{Y} \to \mathbf{X}) = w(\mathbf{Y})T(\mathbf{Y} \to \mathbf{X}) \end{aligned} \tag{7.220}$$

In the first equality we split the transition probability into its two parts as per Eq. (7.217). In the second equality we assumed that $R(\mathbf{X} \to \mathbf{Y}) < 1$, in which case Eq. (7.219) tells us that $\alpha(\mathbf{X} \to \mathbf{Y}) = R(\mathbf{X} \to \mathbf{Y})$; we also took the opportunity to plug in the ratio as per Eq. (7.218). In the third equality we cancelled what we could and also introduced an $\alpha(\mathbf{Y} \to \mathbf{X})$ term: since we assumed that $R(\mathbf{X} \to \mathbf{Y}) < 1$, the definition of R in Eq. (7.218) shows that $R(\mathbf{X} \to \mathbf{Y})R(\mathbf{Y} \to \mathbf{X}) = 1$ so we will have $R(\mathbf{Y} \to \mathbf{X}) > 1$; but in that case, Eq. (7.219) will lead to $\alpha(\mathbf{Y} \to \mathbf{X}) = 1$, meaning we were justified in including this term. The fourth equality identified the two parts of the transition probability as per Eq. (7.217). Thus, we have shown that for the Metropolis–Hastings algorithms, if $R(\mathbf{X} \to \mathbf{Y}) < 1$, the left-hand side of Eq. (7.213) is equal to the right-hand side, i.e., the condition of detailed balance is met. The argument for the case of $R(\mathbf{X} \to \mathbf{Y}) > 1$ is precisely analogous, this time starting from the right-hand side and reaching the left.

In summary, we have shown that *the Metropolis–Hastings algorithm satisfies detailed balance*. As we discussed after Eq. (7.216), obeying detailed balance means that $w(\mathbf{X})$ is a stationary distribution and our random walk will be asymptotically approaching that stationary distribution, i.e., our equilibrium distribution will be $w(\mathbf{X})$. Thus, we have managed to draw random samples from a d-dimensional distribution, which was our goal all along. Once again, take a moment to appreciate how different this all is from what we were doing in earlier sections of this chapter: our $\mathbf{X}_i$'s now form a Markov chain. This means that they are *not statistically independent* from one another. As one of the problems shows, some care must be taken in evaluating averages along the random walk; we implement one possible strategy below, but other approaches also exist, e.g., "binning". Even so, it's important to emphasize that the Metropolis–Hastings algorithm has allowed us to draw random samples from a d-dimensional $w(\mathbf{X})$ simply by calculating ratios of w's (and perhaps also of π's) at two configurations each time. There was no need to worry about a change of variables, no numerical multidimensional inversion, and so on. Simply by evaluating known quantities at a given and a trial configuration, we managed to solve a complicated sampling problem. This is why the Metropolis–Hastings prescription is routinely listed among the most important algorithms of the twentieth century.

How to Implement the Metropolis Algorithm

We now give a practical step-by-step summary of the Metropolis algorithm, which allows us to sample from the d-dimensional distribution $w(\mathbf{X})$.

0. Start at a random location, $\mathbf{X}_0$. It shouldn't matter where you start, since the Markov chain will "equilibrate", reaching the same stationary distribution, anyway. You could account for this "burn-in" time by discarding some early iterations or you could start from an $\mathbf{X}_0$ that is not highly unlikely, i.e., pick $\mathbf{X}_0$ such that $w(\mathbf{X}_0)$ is not too small.
1. Take $\mathbf{X}_{i-1}$ as given (this is $\mathbf{X}_0$ the first time around) and produce a uniformly distributed proposed step according to:
$$\mathbf{Y}_i = \mathbf{X}_{i-1} + \theta \times \mathbf{U}_i \tag{7.221}$$
where $\mathbf{U}_i$ is a d-dimensional sample of uniformly distributed random numbers from -1 to 1; here θ is a number that controls the "step size" and $\mathbf{Y}_i$ is the proposed walker configuration. This step, being uniformly[52] distributed in a multidimensional cube of side 2θ, is *de facto* employing a proposal distribution π that is symmetric; this means that we are actually dealing with the simple Metropolis algorithm (as will also be the case in the Project below). The value of θ is chosen such that roughly 15% to 50% of the proposed steps are accepted.
2. We can combine Eq. (7.218) together with Eq. (7.219). Since we're dealing with a symmetric proposal distribution, the acceptance probability is:
$$\alpha(\mathbf{X}_{i-1} \to \mathbf{Y}_i) = \min\left[1, \frac{w(\mathbf{Y}_i)}{w(\mathbf{X}_{i-1})}\right] \tag{7.222}$$
Only the ratio $w(\mathbf{Y}_i)/w(\mathbf{X}_{i-1})$ matters: even if you don't know w's normalization, you can still carry out the entire process, since the normalization constant cancels.
3. With probability $\alpha(\mathbf{X}_{i-1} \to \mathbf{Y}_i)$, set $\mathbf{X}_i = \mathbf{Y}_i$ (the proposed step is accepted), otherwise set $\mathbf{X}_i = \mathbf{X}_{i-1}$ (the proposed step is rejected). In practice, this is done as follows: generate ξ_i, a random number uniformly distributed from 0 to 1. Then set:
$$\mathbf{X}_i = \begin{cases} \mathbf{Y}_i, & \text{if } \alpha(\mathbf{X}_{i-1} \to \mathbf{Y}_i) \geq \xi_i \\ \mathbf{X}_{i-1}, & \text{if } \alpha(\mathbf{X}_{i-1} \to \mathbf{Y}_i) < \xi_i \end{cases} \tag{7.223}$$
Note that random numbers appear in two distinct roles: first, they allow us to produce the proposed walker configuration, $\mathbf{Y}_i$, as per Eq. (7.221), and, second, they help us decide whether to accept or reject the proposed step as per Eq. (7.223). If $\alpha(\mathbf{X}_{i-1} \to \mathbf{Y}_i) = 1$ the proposed step is accepted regardless of the value of ξ_i. Crucially, the next configuration is determined by the magnitude of $\alpha(\mathbf{X}_{i-1} \to \mathbf{Y}_i)$, which in its turn depends on the value of the ratio $w(\mathbf{Y}_i)/w(\mathbf{X}_{i-1})$; thus, two evaluations (one of which is new) of the (analytically known) desired distribution w are enough to select a new sample.
At this point, you can appreciate our comment about how to pick the value of θ: if θ is tiny, you're taking a small step; since you were most likely already in a region where $w(\mathbf{X}_{i-1})$ was large, it's reasonable to expect that $w(\mathbf{Y}_i)$ will also be large, in which case the step will likely be accepted; but in this case you're not moving very far from where

[52] This is, obviously, not a unique choice; an even more commonly employed step is normal, i.e., Gaussian.

you started, so your sampling is of poor quality. On the other hand, if θ is huge, then you are likely to leave the region of large $w(\mathbf{X}_{i-1})$ and go to a region of small $w(\mathbf{Y}_i)$, in which case the step is likely to be rejected; in this case, too, your sampling is bad, since you are not producing truly new (accepted) samples. By picking θ between these two extremes, you ensure that you can carry out the sampling process efficiently; the value of θ impacts the *acceptance probability* (which, as per Eq. (7.222), refers to a single Metropolis step) and therefore also the *acceptance rate* (how many steps were accepted overall, for the whole calculation). While the lore of computational physics instructs you to aim for an acceptance rate of 50%, statisticians [80] have shown under reasonably general conditions[53] that the optimal acceptance rate is 23.4%. The detailed argument can get complicated, but you shouldn't lose sleep over this: an acceptance rate from 15% to 50% is typically efficient enough.

4. Every n_m steps, make a "measurement", i.e., evaluate $f(\mathbf{X}_i)$. You will need this in order to evaluate the right-hand side of Eq. (7.212); note that, because the Markov chain samples are not statistically independent, we are computing the sample mean (and its variance) using *not* every single sample, but every n_m-th one. This is done in order to eliminate the correlation between the samples.
5. Increment i by 1 and go back to step 1. Terminate the entire process when you've generated sufficiently many samples/measurements ($\mathcal{N}$) that you are comfortable with the variance of the sample mean.

If this all seems too abstract to you, you'll be pleased to hear that we will apply the Metropolis algorithm to a 12-dimensional integral in the Project below. Another way of becoming more comfortable with this algorithm is to implement it for the case of a one-dimensional integral (as the problem set asks you to do): it's always a good idea to first apply a new algorithm to a problem you've solved before, where you know what to expect.

7.8 Project: Variational Quantum Monte Carlo

We will now see how the Metropolis algorithm can help us describe *many-particle quantum-mechanical systems*. This is not the first time we have encountered quantum mechanics: the project in chapter 3 revolved around taking the second derivative of a single-particle wave function, whereas the project in chapter 4 studied several interacting spins. The latter investigation assumed that there were no orbital degrees of freedom and therefore turned into a matrix eigenvalue problem. In the present section we will make the "opposite" assumption, i.e., we will study spinless particles, leaving the fermionic case for the problem set. Note that to fully solve the quantum-mechanical problem for many interacting particles, one would need to solve the *many-particle Schrödinger equation*, an involved task that is at the forefront of modern research; we provide the tools that will help you solve the *single*-particle Schrödinger equation in the next chapter, which studies differential equations. Our goal here will be more humble: we try to produce an *upper bound* on the ground-state

[53] For a Gaussian proposal distribution which, as mentioned in the previous footnote, is the typical choice.

energy of a many-particle quantum-mechanical system; this is a task that can be written down in the form of a many-dimensional integral, allowing us to introduce the method known as *variational Monte Carlo* (VMC).

Assume we are dealing with n_p particles, e.g., for $n_p = 4$ these would be the particles labelled 0, 1, 2, and 3, as usual; their positions would be $\mathbf{r}_0$, $\mathbf{r}_1$, $\mathbf{r}_2$, and $\mathbf{r}_3$. Further assume that we are dealing with n_d-dimensional space, e.g., for $n_d = 3$ we have the usual x, y, and z Cartesian components; the components of the position of the 0th particle are therefore $(\mathbf{r}_0)_x$, $(\mathbf{r}_0)_y$, and $(\mathbf{r}_0)_z$. For n_p particles in n_d dimensions, we have to keep track of $n_p n_d$ components in total, e.g., $4 \times 3 = 12$ components for our running example. Employing the notation used in our earlier Monte Carlo discussion, where $\mathbf{X}$ was a d-dimensional vector, we can bundle the $d = n_p n_d$ components into a vector as follows:

$$\mathbf{X} = \begin{pmatrix} (\mathbf{r}_0)_x & (\mathbf{r}_0)_y & (\mathbf{r}_0)_z & (\mathbf{r}_1)_x & \dots & (\mathbf{r}_{n_p-1})_y & (\mathbf{r}_{n_p-1})_z \end{pmatrix}^T \tag{7.224}$$

One could employ an $n_p \times n_d$ matrix, instead; regardless of how one stores them, the essential point is that there will be $d = n_p n_d$ numbers to keep track of.

7.8.1 Hamiltonian and Wave Function

Our goal will be, in n_d dimensions for n_p particles described by a Hamiltonian $\hat{H}$, to determine the ground-state energy of the system, E_0, reasonably well, using a trial wave function, ψ_T. In general, $\hat{H}$ will be complicated, so the problem will not be amenable to an analytical solution. We will only provide a *variational* answer: our approximation of E_0 will be an upper bound, i.e., is guaranteed to be larger than or equal to the true ground-state energy; we will accomplish this by evaluating a multidimensional integral using techniques from the present chapter. We will now pose a specific problem: four interacting particles in a three-dimensional anisotropic harmonic oscillator.

Hamiltonian

Our chosen four-particle Hamiltonian consists of a (non-relativistic) kinetic energy, a one-body potential energy, and a two-body interaction term:

$$\hat{H} = -\frac{\hbar^2}{2m} \sum_{j=0}^{3} \nabla_j^2 + \frac{1}{2} m \sum_{j=0}^{3} \left\{ \omega_x^2 [(\mathbf{r}_j)_x]^2 + \omega_y^2 [(\mathbf{r}_j)_y]^2 + \omega_z^2 [(\mathbf{r}_j)_z]^2 \right\} + g \sum_{\substack{j,k=0 \\ j<k}}^{3} \exp\left[-(\mathbf{r}_j - \mathbf{r}_k)^2 \right] \tag{7.225}$$

where the particles have equal mass m and ∇_j^2 is the Laplacian taking the second derivative with respect to position $\mathbf{r}_j$. The oscillator strength in each direction is determined by the frequencies ω_x, ω_y, and ω_z, i.e., we are allowing for the possibility of an *anisotropic* oscillator (for which the frequencies are unequal); note the unmistakable quadratic dependence on the position components, identifying this as a *harmonic* oscillator. The last term on the right-hand side is the two-body interaction, which we have taken to be of Gaussian form;

this interaction is characterized by a *strength* parameter g. Note the $j < k$, which ensures that we're not double-counting: if the 0th particle interacts with the 2nd particle, then you shouldn't separately include a term for the 2nd particle interacting with the 0th particle.

Let's spend some time interpreting this Hamiltonian physically. The particles have kinetic energy: they're free to move around. They also have a harmonic-trapping potential energy: this is the result of each individual particle separately interacting with the external field (i.e., the harmonic oscillator). Finally, the two-body interaction term has to do with how strongly the particles attract or repel each other; we will study the case of positive g, so the particles will be repelling each other (i.e., overall increasing the total energy); the two-body term depends on the distance between each pair of particles. You can also consider the Hamiltonian from the perspective afforded by *perturbation theory*: split it into an analytically known term, on the one hand, and a complicated term, on the other, i.e., $\hat{H} = \hat{H}_0 + \hat{H}_1$; in this case, $\hat{H}_0$ would be kinetic plus oscillator (which is the textbook problem we covered in section 3.5) and $\hat{H}_1$ would be the two-body interaction term. Note that, in what follows, we will investigate general (i.e., not necessarily small) values of g.

Trial Wave Function

The split into $\hat{H}_0$ and $\hat{H}_1$ we just mentioned will also guide our choice of the trial wave function. Perhaps we should first explain why this is called a *trial* wave function: the reason is that the full $\hat{H}$ in Eq. (7.225) is quite complicated, so we won't attempt to completely solve the corresponding many-body Schrödinger equation. Instead, we will limit ourselves to employing an *Ansatz*, which is a fancy way of saying "a guess"; we will choose a trial wave function, denoted ψ_T, which will hopefully contain some of the physics involved in our problem. As should be clear by this discussion, this is a never-ending process: unless you can prove (via another method) that your wave function is the true ground state (and therefore your estimate of the ground-state energy E_0 is *exact*), you will always be able to go back to the drawing board and design a new, hopefully better, trial wave function; the problem set invites you to do better than we will with our choice below. Crucially, this is not a directionless endeavor: since we will be obeying a *variational* principle, a trial wave function that leads to a lower energy is *better* (i.e., closer to the, unknown, true ground-state energy). This then becomes a practical question: if your new guesses no longer lead to a marked improvement, then you can call it a day.

We won't have to worry about the particles' spin; this could mean either *bosons* or *boltzmannons*, i.e., either indistinguishable particles obeying Bose–Einstein statistics, or distinguishable particles obeying Maxwell–Boltzmann statistics.[54] In that case, we can place each of the four particles into the same single-particle state ϕ:

$$\psi_T(\mathbf{X}) = \phi(\mathbf{r}_0)\phi(\mathbf{r}_1)\phi(\mathbf{r}_2)\phi(\mathbf{r}_3) \tag{7.226}$$

[54] This is merely the distinction between quantum *mechanics* and quantum *statistics*. Imagine neutrons and protons with different spin projections, or ultracold atoms belonging to distinct species.

On the left-hand side we are using our Monte-Carlo-esque notation from Eq. (7.224); obviously, our four-particle state is merely the product of four copies of the same single-particle state. We stress that we are *allowed* to place all four particles in the same ϕ.

We pick this ϕ motivated by the split into $\hat{H}_0$ and $\hat{H}_1$: imagine for a moment that $g = 0$, i.e., we didn't have a two-body interaction term; the resulting Hamiltonian is then simply the sum of four single-particle Hamiltonians. That means that our study in section 3.5 carries over; of course, there we were faced with a single particle in a single dimension, so we first have to generalize to the three-dimensional case. The problems for each of the three Cartesian components are totally decoupled, so ϕ can simply be another product, of the wave functions solving those three problems:

$$\phi(\mathbf{r}_0) = \varphi_{n_x}((\mathbf{r}_0)_x)\,\varphi_{n_y}((\mathbf{r}_0)_y)\,\varphi_{n_z}((\mathbf{r}_0)_z) \tag{7.227}$$

where we chose the 0th particle for concreteness. Here the φ_{n_x}, φ_{n_y}, and φ_{n_z} have the form of the one-dimensional solution in Eq. (3.72); be sure to distinguish between ϕ and φ. We reiterate that there are two products involved here: Eq. (7.226) is a product over particle labels and Eq. (7.227) a product over Cartesian components for a given particle label. Note also that each Cartesian component gets its own quantum number; since we are studying the ground-state of the system, we are justified in choosing each of these to be zero, i.e., $n_x = n_y = n_z = 0$. Similarly to what we did in Code 3.3, i.e., `psis.py`, we also allow for the possibility of including an extra variational parameter, α, in the exponent of the Gaussian, leading to:

$$\varphi_0\left((\mathbf{r}_0)_x\right) = e^{-\alpha m\omega_x[(\mathbf{r}_0)_x]^2/(2\hbar)} \tag{7.228}$$

and similarly for the y and z components. Note that if you take the two-particle interaction (i.e., g in Eq. (7.225)) to be strong, you can find a minimizing α that is not the "obvious" one ($\alpha = 1$); we will see this explicitly below.

Putting the last three equations together, we are led to a many-particle wave function that is a product of 12 terms: four ϕ's, each of which is a product of three φ's. For our specific case, we can further manipulate the product to write it in terms of a sum in the exponent. Note that this form is not set in stone: our steps *motivate* the wave function, but do not derive it: one could envision more complicated choices being made (e.g., instead of a single α, having different variational parameters in each direction).

Implementation

Before we proceed to develop the variational principle and the VMC method, let's implement what we already have, namely the trial many-particle wave function of Eq. (7.226) and the effect the Hamiltonian of Eq. (7.225) has on it. We now wish to quantify the latter; we recall that in Eq. (3.83) we had defined the local kinetic energy. We generalize that to the following definition of the *local energy*:

$$E_L(\mathbf{X}) = \frac{\hat{H}\psi_T(\mathbf{X})}{\psi_T(\mathbf{X})} \tag{7.229}$$

eloc.py Code 7.6

```
import numpy as np

def psi(al, oms, rs):
     rexp = np.sum(oms*rs**2)
     return np.exp(-0.5*al*rexp)

def ekin(al, oms, rs, h=0.01, hom = 1.):
     npart, ndim = rs.shape
     psiold = psi(al, oms, rs)
     kin = 0.
     for j in range(npart):
          numer = 0.
          for el in range(ndim):
               r = rs[j,el]
               rs[j,el] = r + h
               psip = psi(al, oms, rs)
               rs[j,el] = r - h
               psim = psi(al, oms, rs)
               rs[j,el] = r
               numer += psip + psim - 2.*psiold
          lapl = numer/h**2
          kin += -0.5*hom*lapl/psiold
     return kin

def epot(oms, rs, strength = 3, m = 1.):
     npart, ndim = rs.shape
     pot = 0.5*m*np.sum(oms**2*rs**2)
     for k in range(1,npart):
          for j in range(k):
               r2 = np.sum((rs[j,:] - rs[k,:])**2)
               pot += strength*np.exp(-r2)
     return pot

if __name__ == '__main__':
     npart, ndim, al = 4, 3, 0.6
     oms = np.arange(1, 1 + ndim)
     rs = np.arange(1, 1 + npart*ndim).reshape(npart, ndim)
     rs = 1/rs
     print(ekin(al, oms, rs), epot(oms, rs))
```

Since this involves the full Hamiltonian, it is the local *total* energy. It's called local because it depends on the specific "walker" $\mathbf{X}$ you are at; in other words, it's something you can evaluate once you've placed all the particles at given locations, as per Eq. (7.224). In the code, we will split this local energy into two parts, by dividing the full Hamiltonian into kinetic plus potential energy, $\hat{H} = \hat{T} + \hat{V}$; note that this is different from the earlier split $\hat{H} = \hat{H}_0 + \hat{H}_1$, which was designed to capture as much as possible of the interactions in a form that can be analytically handled; our latest split is simply bookkeeping: we separate the kinetic energy (which, for a non-relativistic system, is always going to be the same) from the potential energy (which depends on the specific problem).

Code 7.6 starts by defining `psi()`, a Python function that evaluates the wave function of the many-particle system for a given variational parameter (α), a given set of oscillator frequencies ($\omega_x, \omega_y, \omega_z$), and a given set of particle positions (which bundles together all the $\mathbf{r}_j$'s). This is an implementation of Eq. (7.226), Eq. (7.227), and Eq. (7.228), one plugged into the other. Our example employs broadcasting of a 1d array with a 2d array, which works in our case because the dimensions match: for our example of four particles in three dimensions, `oms` has three components and `rs` is a 4×3 matrix. If you are not comfortable with this, you should write out the wave function evaluation the "naive" way. Note that the code is quite general: if we wanted to study, say, 11 particles in two dimensions, `psi()` would work equally well. We start this program with the wave function implementation, because the next function (`ekin()`, for the local kinetic energy) uses `psi()` without receiving it as a parameter. However, it's crucial to note that `ekin()` is general and will work for a different wave function as soon as you change one or two lines in the body of `psi()`.

The function `ekin()` accepts the same parameters as `psi()`, as well as two other ones (for the step size h and for the value of $\hbar^2/m$) which are given default values. We first extract the numbers of particles and dimensions, based on the position matrix that was passed in: once again, this is all general and would work just as well for other problems. The core of this function is very similar to code 3.4, i.e., `kinetic.py` from chapter 3; this time around, in addition to studying a many-particle problem, we are also employing `numpy` arrays instead of Python lists. Thus, we carry out one wave function evaluation at the start and then six evaluations for each particle; this is simply the finite-difference approximation to the Laplacians in Eq. (7.225). We are intentionally not being very Pythonic, since we choose to iterate over indices explicitly. We use j for the particle index and l for the Cartesian-component index. Once again, we first read out the value of the element, modify it by either adding or subtracting the step size, and then restore its original value at the end. It's important to emphasize that `ekin()` doesn't depend on details of the wave function; if you take the second derivatives analytically (as the problem set asks you to do) you will avoid these $6n_p + 1$ wave function evaluations (more generally: $2n_d n_p + 1$ evaluations), thereby producing a more efficient but less general code. As a matter of fact, `ekin()` doesn't even depend on details of the Hamiltonian in Eq. (7.225): even if you wanted to study a different problem in the future (say, particles in a periodic box with no one-body potential and a different two-body potential) the kinetic energy (function) would stay the same.

The function `epot()` is a straightforward implementation, corresponding to the one- and two-body interaction terms in the Hamiltonian of Eq. (7.225). We mentioned above

that `ekin()` doesn't depend on the details of the Hamiltonian, since the kinetic energy is always the same; `epot()` *does* depend on the details of the Hamiltonian, but it encapsulates them, i.e., if you wish to use different interaction terms in the future you would have to touch only these lines of code. Just like in `ekin()`, we start out by extracting the number of particles and number of dimensions, meaning that everything that follows will be quite general. The implementation of the one-body term uses broadcasting, just as we saw in our discussion of `psi()`. The implementation of the two-body term imposes the $j < k$ condition in the summation by having the two indices take the values $k = 1, 2, \ldots, n_p - 1$ and $j = 0, 1, \ldots, k - 1$. It uses `numpy` array slicing to select $\mathbf{r}_j$ and $\mathbf{r}_k$, and then carries out the squaring and summing of the components automatically. It's worth noting that we have picked the default value of the strength g to be large; this will allow us, below, to investigate a case that is not an already-solved problem. Note, finally, that `epot()` does not take in α as a parameter; as a matter of fact, it doesn't even call the function `psi()`; from the definition in Eq. (7.229) we see that for local interaction terms the wave function cancels: $\hat{V}\psi_T(\mathbf{X})/\psi_T(\mathbf{X}) = V(\mathbf{X})$.

The main program assigns specific values to our input parameters and then makes up an arbitrary configuration of particle positions, merely as a test case. It then calls the kinetic and potential functions, which evaluate the energies *for that one configuration of particles*. The output of this code is not very important, since there is no deep physical meaning behind the specific choice of where the particles are located. Incidentally, this entire program does not involve *any* random numbers: for a given configuration, we can deterministically evaluate the wave function, the local kinetic energy, and the local potential energy. There is nothing stochastic going on so far: in the following section we'll see how to combine these functions together with Monte Carlo integration to approximate the ground-state energy of the system (i.e., not the local energy for a single walker).

If you are looking to build some intuition around the magnitude of the numbers involved, you may wish to turn off the two-body interaction (i.e., set $g = 0$) as a test case: if you also take $\alpha = 1$, then you can analytically calculate the energy of the system. Since there are no two-body interactions in this scenario, we are dealing with four independent particles, so we can focus on a single one. Then, the energy of a single particle in a given dimension, say y, is simply $(n_y + 0.5)\hbar\omega_y$ as per Eq. (3.71); for the specific set of frequencies we chose (given also that $n_x = n_y = n_z = 0$), we therefore expect each particle to contribute an energy of $0.5 + 0.5 \times 2 + 0.5 \times 3 = 3$, leading to a value of 12 in total; we expect to find this *regardless* of the specific positions the particles are placed in (here and below, all energies are given in units of $\hbar\omega_x$). Of course, the function `ekin()` in Code 7.6 involves a default step size,[55] `h`, which may lead to a numerical error; for our example walker `rs` the code gives `11.9994839168`, which is showing that things are behaving as expected.

7.8.2 Variational Method

Code 7.6 evaluates the wave function and the local kinetic and local potential energies, but doesn't tell you how to pick the particle positions. The $\mathbf{r}_j$'s have to be selected somehow;

[55] The finite-difference approximation to the Laplacian employs a "step size" h, but we also encountered another "step size" θ in Eq. (7.221); the latter refers to the steps carried out in the d-dimensional space.

in the present section, we see how to pick them stochastically, in a way that allows us to approximate the ground-state energy.

Rayleigh–Ritz Principle

We now go over a very important result from elementary quantum mechanics: this is variously known as the *Rayleigh–Ritz principle* or the *variational principle*. In words, it says that the (normalized) expectation value of the Hamiltonian $\hat{H}$ evaluated with a trial wave function ψ_T is an upper bound on the true ground-state energy E_0. If you've encountered this principle in a course on quantum mechanics, this was most likely in the context of single-particle wave functions; however, as you'll soon see, the derivation and result are fully general, which means that we'll be able to apply this principle also in the context of *many*-particle quantum mechanics.

Take the complete set of orthonormal eigenstates $|u_n\rangle$ of the Hamiltonian $\hat{H}$:

$$\hat{H}|u_n\rangle = E_n|u_n\rangle \tag{7.230}$$

Note that this is "diagonal", since the $|u_n\rangle$ are the exact energy eigenstates. The corresponding eigenvalues are E_n and are ordered such that $E_0 \leq E_1 \leq \ldots$; thus, the ground state has the state vector $|u_0\rangle$ and energy E_0. Of course we don't actually know these: if we did, then we would have already solved a major problem in many-particle quantum mechanics. Even so, we are free to expand our trial wave function (state vector) $|\psi_T\rangle$ in terms of the (unknown) exact eigenstates $|u_n\rangle$:

$$|\psi_T\rangle = \sum_{n=0}^{\infty} \gamma_n |u_n\rangle \tag{7.231}$$

which is a step that is frequently carried out in quantum mechanics courses.[56]

We now decide to define the following *variational energy* E_V:

$$E_V = \frac{\langle\psi_T|\hat{H}|\psi_T\rangle}{\langle\psi_T|\psi_T\rangle} \tag{7.232}$$

and then write the kets and bras as per Eq. (7.231):

$$E_V = \frac{\sum_{n=0}^{\infty}\sum_{n'=0}^{\infty}\gamma_n\gamma_{n'}^*\langle u_{n'}|\hat{H}|u_n\rangle}{\sum_{n=0}^{\infty}\sum_{n'=0}^{\infty}\gamma_n\gamma_{n'}^*\langle u_{n'}|u_n\rangle} = \frac{\sum_{n=0}^{\infty}\sum_{n'=0}^{\infty}\gamma_n\gamma_{n'}^* E_n\langle u_{n'}|u_n\rangle}{\sum_{n=0}^{\infty}\sum_{n'=0}^{\infty}\gamma_n\gamma_{n'}^*\langle u_{n'}|u_n\rangle} = \frac{\sum_{n=0}^{\infty}|\gamma_n|^2 E_n}{\sum_{n=0}^{\infty}|\gamma_n|^2} \tag{7.233}$$

In the second step we took advantage of the fact that we know what effect $\hat{H}$ has on the energy eigenstates, as per Eq. (7.230). In the third step we employed the orthonormality of the exact eigenstates (in both the numerator and the denominator) to get rid of the sum(s) over n'. We now write $E_n = E_0 + E_n - E_0$ in the numerator to find:

$$E_V = E_0 + \frac{\sum_{n=1}^{\infty}|\gamma_n|^2(E_n - E_0)}{\sum_{n=0}^{\infty}|\gamma_n|^2} \geq E_0 \tag{7.234}$$

[56] Another way of looking at this relation is by taking $|\psi_T\rangle$ and inserting a resolution of the identity, i.e., $|\psi_T\rangle = \sum_{n=0}^{\infty}|u_n\rangle\langle u_n|\psi_T\rangle$, from which we immediately see that $\gamma_n = \langle u_n|\psi_T\rangle$.

where the last step follows from the positivity of $|\gamma_n|^2$ and of $E_n - E_0$; you can take the sum in the numerator to start at $n = 0$ if you wish, but that term would give zero anyway.

In short, we have the foundation of the *Rayleigh–Ritz (or variational) principle*:

$$E_V = \frac{\langle\psi_T|\hat{H}|\psi_T\rangle}{\langle\psi_T|\psi_T\rangle} \geq E_0 \tag{7.235}$$

As advertised, this says that by forming the expectation value of the Hamiltonian with respect to a trial wave function you produce an upper bound on the true ground-state energy. When ψ_T is the exact ground-state wave function, then we reach the equality in this relation (i.e., we get the exact ground-state energy).[57] Of course, since the true ground state is actually unknown, what we do, instead, is write down ψ_T in terms of one or more variational[58] parameters, like α in Eq. (7.228). Thus, $\psi_T = \psi_T(\alpha)$ leads to a variational energy which also depends on the value of α, i.e., $E_V = E_V(\alpha)$. By minimizing $E_V(\alpha)$ we will therefore try to get as close to E_0 as we can. As promised, none of these steps or conclusions have anything to do with single-particle quantum mechanics; they are completely general.

Variational Monte Carlo

Armed with the Rayleigh–Ritz principle, we will now see how to practically evaluate the variational E_V; if you have a way of accomplishing that task, then you can also minimize E_V to get a good upper bound on the ground-state energy.

Start by taking the variational energy from Eq. (7.235); this time, instead of expanding in terms of the exact eigenstates like in Eq. (7.231), write out the matrix elements in terms of integrals in coordinate space:[59]

$$E_V = \frac{\int d^d x\, \psi_T^*(\mathbf{x})\hat{H}\psi_T(\mathbf{x})}{\int d^d x\, \psi_T^*(\mathbf{x})\psi_T(\mathbf{x})} = \frac{\int d^d x\, |\psi_T(\mathbf{x})|^2 \frac{\hat{H}\psi_T(\mathbf{x})}{\psi_T(\mathbf{x})}}{\int d^d x\, |\psi_T(\mathbf{x})|^2} = \int d^d x \left(\frac{|\psi_T(\mathbf{x})|^2}{\int d^d x\, |\psi_T(\mathbf{x})|^2} \right) \frac{\hat{H}\psi_T(\mathbf{x})}{\psi_T(\mathbf{x})} \tag{7.236}$$

where $\mathbf{x}$ is a d-dimensional entity. In the second step we multiplied and divided with the wave function (in the numerator). In the third step we simply regrouped terms.

Our final result is now of a familiar form: it looks exactly like the $\int w(\mathbf{x}) f(\mathbf{x}) d^d x$ in Eq. (7.212) for multidimensional weighted Monte Carlo integration, if we identify:

$$w(\mathbf{x}) = \frac{|\psi_T(\mathbf{x})|^2}{\int d^d x\, |\psi_T(\mathbf{x})|^2}, \qquad f(\mathbf{x}) = \frac{\hat{H}\psi_T(\mathbf{x})}{\psi_T(\mathbf{x})} \tag{7.237}$$

with the latter being none other than the *local energy* of Eq. (7.229). In words, we have mapped the problem of computing the variational energy E_V into a multidimensional integral with the weight being controlled by $|\psi_T(\mathbf{x})|^2$ and the function we are integrating being the local energy $E_L(\mathbf{x})$. This is probably a good time to note that the variational energy E_V

[57] You will explicitly show this in the problem set.

[58] We can finally see why this parameter was called "variational" all this while.

[59] Imagine inserting resolutions of the identity, this time in terms of position eigenstates: $\int d^d x |\mathbf{x}\rangle\langle\mathbf{x}|$.

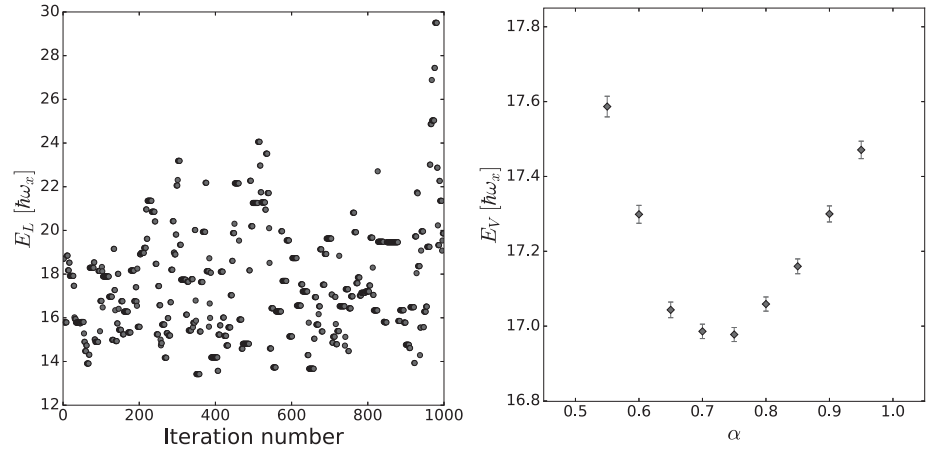


Fig. 7.8 Consecutive local energies for $\alpha = 0.6$ (left), variational minimization in action (right)

does *not* depend on the positions of the particles, since it integrates over all the $\mathbf{x}$'s; this makes it markedly different from the local energy E_L, which is a function of $\mathbf{x}$.

Given our mapping, we now have a prescription in Eq. (7.212) regarding how to produce the sample mean and its standard deviation: draw samples from $w(\mathbf{x})$ and use them to evaluate the local energy. The only complication is how to produce d-dimensional variates drawn from $w(\mathbf{x})$; but that, too, is a task we've already accomplished, via the Metropolis algorithm, see Eq. (7.221) and below. As per Eq. (7.222), the acceptance probability is determined by evaluating the ratio of the weight function at the proposed and current walker configurations; from Eq. (7.237) we see that this is simply the ratio of the squares of the moduli of the trial wave functions (the denominators cancel). Using this acceptance probability in the Metropolis algorithm will lead to a random walk that is asymptotically approaching the desired weight distribution. To reiterate:[60]

$$E_V \approx \frac{1}{\mathcal{N}} \sum_{i=0}^{\mathcal{N}-1} E_L(\mathbf{X}_i) \pm \frac{1}{\sqrt{\mathcal{N}-1}} \sqrt{\frac{1}{\mathcal{N}} \sum_{i=0}^{\mathcal{N}-1} E_L^2(\mathbf{X}_i) - \left[\frac{1}{\mathcal{N}} \sum_{i=0}^{\mathcal{N}-1} E_L(\mathbf{X}_i) \right]^2} \tag{7.238}$$

This is merely Eq. (7.212) cast in the language of our problem; the $\mathbf{X}_i$'s are drawn from $|\psi_T(\mathbf{x})|^2$ via the Metropolis algorithm.

At this point, let us return to Step 4 in our suggested implementation of the Metropolis algorithm: this involved making a "measurement", i.e., an evaluation of E_L only every n_m steps, whereas the evaluation of the $|\psi_T|^2$ ratio *has* to be carried out at every step in order to make a decision. In order to see why, imagine making several hundred Metropolis steps using a weight of $|\psi_T|^2$ and evaluating E_L at every single one of them (i.e., with $n_m = 1$): the result is shown in the left panel of Fig. 7.8, for the same input parameters as in Code 7.6.

[60] As mentioned near Eq. (7.212), we dropped the V because we're interested in integrating over all of space; note that this V has nothing to do with the potential energy, $V(\mathbf{X})$.

It is in the nature of our algorithm to start from a given walker configuration and produce a new one by making a small change (a step): as a result, the configurations making up our Markov chain are clearly correlated. This can be seen in the figure from the fact that several of the circles are overlapping, implying that very small steps can and do get accepted; if you make θ smaller, you will encounter even more features of large-scale structure. You can see, especially for the last 50 iterations of the figure, that if your walker gets "stuck" in a region where the energy is unrealistically large, it takes a while to climb down to more reasonable values. Our practical prescription is to only measure the local energy every once in a while, letting the Markov chain de-correlate in the meantime; in the code that we are about to introduce, the E_L's for the 1000 consecutive iterations (being produced by 1000 evaluations of the $|\psi_T|^2$ ratio in the background) shown in the left panel of Fig. 7.8 will be replaced by 10 E_L evaluations. Note that Eq. (7.238) involves $\mathcal{N}$ evaluations of the local energy: since we are making measurements every n_m steps, this means we'll need $n_m\mathcal{N}$ Metropolis steps in total. The problem set studies these issues in more detail.

Implementation

Code 7.7 starts out by importing the wave function, the local kinetic and local potential energy functions from Code 7.6 and the statistics function from Code 7.5; note that the latter is a one-dimensional code, but `stats()` works just as well here, since the evaluation of the sample mean and its standard deviation in Eq. (7.238) is fully analogous to that in Eq. (7.193); our walker is sampling a d-dimensional distribution and, similarly, the local energy is a function of a d-dimensional variable, but at the end of the day we're carrying out the same basic statistical operations (mean and variance of a scalar). Having already built all the necessary infrastructure, our VMC code is quite simple and is placed in a single function, `vmc()`; our earlier point about `stats()` working in both one-dimensional and d-dimensional settings has an analogue here: `vmc()` is written for the d-dimensional case, manipulating walkers that are matrices with $n_p \times n_d$ elements, but it is quite similar to the one-dimensional Metropolis code that you are asked to develop in the problem set.

We now turn to a detailed discussion of `vmc()`. There is no parameter for the particle positions, since `vmc()` will generate the walker configurations itself; thus, the particle number (n_p) and number of dimensions (n_d) have to be explicitly passed in this time. As in earlier functions, the next two parameters correspond to the variational parameter (α) and the oscillator frequencies ($\omega_x, \omega_y, \omega_z$). The final parameter is the seed that gets the random number sequence going, given a default value here. The first line in the body of `vmc()` sets up three other quantities: the total number of evaluations of the local energy to be carried out ($\mathcal{N}$), the number controlling how often these evaluations should be carried out (n_m), and the quantity controlling the size of the proposed step in Eq. (7.221), namely θ; our chosen value for n_m is large enough that we are likely removing the correlations (but perhaps are also being a little wasteful). We then pass the seed into `numpy.random.seed()`: note that we employ `numpy` functionality instead of the standard Python one, as discussed at the end of section 7.7.1. We then use `numpy.random.uniform()` to produce the initial (arbitrary) walker configuration, $\mathbf{X}_0$: marvel at how convenient this is, generating an entire matrix of

Code 7.7 vmc.py

```
from eloc import psi, ekin, epot
from montecarlo import stats
import numpy as np

def vmc(npart, ndim, al, oms, inseed=8735):
     Ncal, nm, th = 10**4, 100, 0.8
     np.random.seed(inseed)
     rolds = np.random.uniform(-1, 1, (npart, ndim))
     psiold = psi(al, oms, rolds)
     iacc, imeas = 0, 0
     eners = np.zeros(Ncal)

     for itot in range(nm*Ncal):
         rnews = rolds+th*np.random.uniform(-1,1,(npart, ndim))
         psinew = psi(al, oms, rnews)
         psiratio = (psinew/psiold)**2

         if psiratio >= np.random.uniform(0,1):
             rolds = np.copy(rnews)
             psiold = psinew
             iacc +=1
         if (itot%nm)==0:
             eners[imeas] = ekin(al,oms,rolds)+epot(oms,rolds)
             imeas += 1

     return iacc/(nm*Ncal), eners

if __name__ == '__main__':
     npart, ndim, al = 4, 3, 0.6
     oms = np.arange(1, 1 + ndim)
     accrate, eners = vmc(npart, ndim, al, oms)
     av, err = stats(eners)
     print(accrate, av, err)
```

uniformly distributed numbers (with the dimensions we need) in one line of code. Later in the program, `numpy.random.uniform()` is used twice more: first, when making a proposed step; this is what Eq. (7.221) called $\mathbf{U}_i$ which, you may recall, is a d-dimensional entity. Second, we also need the random number ξ_i as per Eq. (7.223); for this case we

make use of the fact that if you don't pass a third argument to `numpy.random.uniform()` then it returns a single number.

It's important to keep straight the different integers involved here: n_p, n_d, $\mathcal{N}$, and n_m are kept fixed for a given VMC run, i.e., they are "input" in a way. Knowing $\mathcal{N}$ ahead of time allows us to set up a numpy array to keep track of the $E_L(\mathbf{X}_i)$'s that we will be evaluating. In addition to these four constants, our function makes use of three more counters, `itot`, `iacc`, and `imeas`. Of these, `itot` is the index that keeps track of every single Metropolis step (i.e., plays the role of i in Eq. (7.223)); the total number of such steps is $n_m\mathcal{N}$, as discussed earlier. Next, we have `iacc` which is an index we use to keep track of how many proposed steps have been accepted so far; this allows us to return the acceptance rate when the function is done, computed via `iacc/(nm*Ncal)`. Finally, we also have `imeas`, an index we use to store our energy measurements: `itot` goes from 0 to $n_m\mathcal{N} - 1$ and we evaluate the local energy every n_m-th time; when we do so, we increment `imeas` by 1, so we can write the local energy into `eners` the next time we need to make a measurement.

Inside the loop over `itot`, we evaluate the square of the ratio of the trial wave function at the new and old walker configurations: this is what the Metropolis algorithm of Eq. (7.222) requires us to do for a weight function of $|\psi_T(\mathbf{x})|^2$, as mentioned near Eq. (7.238). Instead of the $\mathbf{X}_{i-1}$, $\mathbf{X}_i$, and $\mathbf{Y}_i$ involved in Eq. (7.222) and Eq. (7.223) we here need only two walker variables, `rolds` and `rnews`. When a proposed step is accepted, we use `numpy.copy()` to update `rolds`; we could have used a simple assignment, but we wish to instill best practices, lest you get over-confident with re-assigning `numpy` arrays. If the step gets rejected, we don't have to explicitly do anything (i.e., there's no need for an `else:`), since `rolds` simply keeps the value it already had. Finally, when it's time to make a measurement we call our earlier functions `ekin()` and `epot()` and store their sum into `eners`.

It's worth observing that when we evaluate a proposed step in `vmc()`, we move all the particles at once, as per Eq. (7.221); there exist alternative Monte Carlo algorithms that make "local updates" instead, which in our case could translate to moving one particle at a time (before evaluating a new Metropolis ratio). Another scenario is to employ "heat-bath updates" whereby a few variables are brought to equilibrium, while all the other ones are kept fixed; this is most commonly used in lattice Monte Carlo approaches, a discussion of which would take us too far afield. What we have here is enough for our purposes.

The main program sets up the input parameters, calls `vmc()` to produce the acceptance rate and array of local energies, and then passes the latter to `stats` to compute the sample mean and its standard deviation. If you repeat the entire exercise for many α's, you will find the results shown in the right panel of Fig. 7.8; we have chosen to tune θ to keep the acceptance ratio reasonably stable and have also decided to use different seeds for different α's. The variational energy E_V exhibits a clear minimum at $\alpha \approx 0.75$; crucially, $\alpha = 1$ most certainly is *not* the minimum, meaning that our variational parameter has helped us produce a (small) decrease in the energy. If the specific numbers shown in the right panel of Fig. 7.8 don't mean much to you, you may choose to turn off the two-body interaction (i.e., set $g = 0$) and also take $\alpha = 1$ as a test case: the energy is 12, so that's what we expect our VMC run to produce *for every sample*! If you carry out the test explicitly you therefore find a tiny standard deviation.

It's worth taking a moment to appreciate the significance of our achievement: we were

dealing with a strongly interacting system of four particles in three dimensions, so this isn't really a problem we would have attempted to solve without the use of a computer; we were able to produce an upper bound to the ground-state energy for this system, by evaluating 12-dimensional integrals via the Metropolis algorithm. Equally important is the fact that we've kept the code sufficiently general to allow us to study in the future different particle numbers, different dimensionalities, and different interactions.

It's hard to avoid noticing that the right panel of Fig. 7.8 is minimizing by checking different values of α "by hand". You could envision using one of the minimization techniques from section 5.5, instead of such a manual process; it's important to realize that in the present case our energy estimates are always associated with a *statistical error*, so minimizing the sample mean is not enough. Of course, it's always possible to use both approaches in succession: use an automated minimizer to find the ballpark region where you should then investigate in more detail manually.

7.9 Problems

1. We will see how to evaluate the following Gaussian integral analytically:

$$I = \int_{-\infty}^{\infty} dx e^{-x^2/2} \tag{7.239}$$

   The main trick is to first look at the square of the integral:

$$I^2 = \int_{-\infty}^{\infty} dx e^{-x^2/2} \int_{-\infty}^{\infty} dy e^{-y^2/2} \tag{7.240}$$

   where we used a new name for the second integration variable. The next step is to go from Cartesian coordinates, x and y, to polar coordinates, r and θ. It should be straightforward from there. Make sure to take the square root at the end of the entire process.
2. Carry out an error analysis for the midpoint rule (Taylor expand around $x_i + h/2$ and keep terms up to second order). This should include both the one-panel case and the composite case. Then, implement the midpoint rule in Python for the usual case of $f(x) = 1/\sqrt{x^2+1}$ ($a = 0$, $b = 1$) for $n = 51$ and compare with the analytical answer, as well as with the other methods.
3. Take $f(x) = 1$ in Eq. (7.26) and thereby show that, for a Newton–Cotes integration rule, the sum of all the weights is equal to the width of the elementary interval.
4. Re-derive Simpson's rule for an elementary interval, Eq. (7.45), the way you would if you didn't know about Lagrange interpolation: take $f(x) \approx p(x) = \alpha x^2 + \beta x + \gamma$ and then integrate it from x_i to x_{i+2}. Then, enforce the fact that the quadratic passes through the points $f(x_i)$, $f(x_{i+1})$, and $f(x_{i+2})$. You could now try to solve these three equations for the three unknowns α, β, and γ. Instead, take a linear combination of these three equations that gives the same answer as the result of integrating $p(x)$ earlier in this problem; you have thereby arrived at the two-panel Simpson's rule.
5. In the main text we computed the leading error in the one-panel version of the trapezoid rule starting from the Lagrange-interpolation error formula. You will now re-derive

it using a Taylor expansion. Specifically, carry out a Taylor series expansion of $f(x)$ around x_i, keeping terms up to second order. Re-express the first derivative using the forward-difference approximation, Eq. (3.8), starting at the point x_i. Integrate your equation for $f(x)$ from x_i to x_{i+1} and compare the result with the sum of Eq. (7.30) and Eq. (7.35).

6. In the main text we derived the error of the trapezoid rule beyond the leading term, Eq. (7.41), by making use of Eq. (7.38). Here we see another way to arrive at the same result. Specifically, write down two Taylor expansions: first, expand $f(x)$ around x_i and, second, expand $f(x)$ around x_{i+1}. Add the two equations together and divide with 2. Integrate from x_i to x_{i+1} and then add together all the one-panel equations. Observe that the even-order derivative terms can be expressed as composite trapezoid-rule approximations to integrals, e.g., the term proportional to h^3 is an approximation to $(h^2/6)\int_a^b f''(x)dx$; note that these integrals can be trivially carried out. To find the error in *those* terms, simply repeat the whole exercise, using $f''(x)$ instead of $f(x)$, and so on.
7. In the main text we carried out the derivation for the error in Simpson's rule using a Taylor expansion, see Eq. (7.48) and below. We now examine another derivation for the same quantity, this time starting from the interpolation formula Eq. (6.38). In essence, we need to carry out a more complicated version of the integral in Eq. (7.34); the important difference is that the integrand changes sign and therefore you cannot immediately employ the mean-value theorem (if you could pull out the derivative term, the remaining integral would give zero, which is obviously wrong). Instead, employ:

$$\mathcal{E}_i = \frac{1}{4!} f^{(4)}(\eta) \int_{x_i}^{x_{i+2}} (x - x_i)(x - x_{i+1})^2 (x - x_{i+2}) dx \tag{7.241}$$

   The middle term is now squared. Carry out the integral and compare with Eq. (7.52).
8. By repeating the argument of section 7.3.1 (twice), prove Eq. (7.63) and Eq. (7.64).
9. Starting from the elementary-interval expressions for Simpson's 3/8 rule and for Boole's rule given in Table 7.1, implement the composite versions of these approaches. Then, generalize the derivation of section 7.3.1 to cover Boole's rule and then modify our code in `adaptive.py` so that it also works for this case.
10. In `adaptive.py` we gave a general adaptive integration routine. Note, however, that this was wasteful: when you double the number of panels from N to $N' = 2N$, you don't need to throw away all the function values that you've already computed. Explicitly write out the first few cases (I_1, I_2, I_4, I_8) to convince yourself that:

$$I_{N'} = \frac{1}{2} I_N + h_{N'} \sum_{k=1,3,\ldots}^{N'-1} f\left(a + k h_{N'}\right) \tag{7.242}$$

    Implement this approach to produce an adaptive trapezoid integrator that is efficient.
11. Observe that the second column in the output of `romberg.py` is identical to the first few lines in the output for Simpson's rule from `adaptive.py`. Motivated by this fact, apply Eq. (7.88) to the case of $j = 1$, i.e., for $R_{i,1}$. You immediately see that this is expressed in terms of $R_{i,0}$ and $R_{i-1,0}$. Both of these are from the first column, so they are composite-trapezoid rule results; write them out in terms of function evaluations, use

the resulting expressions to do the same for $R_{i,1}$, and then compare the final result with the composite Simpson's rule, Eq. (7.46).

12. In an earlier problem, you produced an efficient (i.e., non-wasteful) adaptive integrator. Do the same for Romberg integration, i.e., rewrite `romberg.py` such that you don't throw away earlier function evaluations when you go from N to $N' = 2N$ panels.
13. Apply Eq. (7.89) to the case of $n = 1$, i.e., demand that you can exactly integrate monomials up to order $2n - 1 = 1$. Find c_0 and x_0 and compare the resulting formula with the one-panel version of the midpoint rule, Eq. (7.20).
14. Show that the monic orthogonal polynomial of degree n is unique. To do so, employ proof by contradiction, i.e., assume that there are two monic orthogonal polynomials, $p_n^A(x)$ and $p_n^B(x)$; their difference is a polynomial of degree $n - 1$ and is therefore orthogonal to both $p_n^A(x)$ and $p_n^B(x)$. Write this out and figure out what it means for the relationship between these two monic polynomials.
15. In order to derive the Christoffel–Darboux identity for Legendre polynomials, given in Eq. (7.107), take the recurrence relation in Eq. (2.86) and multiply with $P_j(y)$. Rewrite your equation with x interchanged with y and then subtract these two equations from each other. If you sum the result for j from 0 to n, you find a telescoping series, i.e., the middle terms cancel. You should be left with Eq. (7.107).
16. If you compute (or look up) the weights for high-order Newton–Cotes rules, you will see that some of these are negative. You will now show that for Gauss–Legendre quadrature the weights are always positive. To see this, plug Eq. (6.48) into Eq. (7.97), split the resulting expression into two integrals, and notice that the second one vanishes. The first one is a definite integral of a non-negative function (and is therefore also non-negative).
17. For Gauss–Legendre quadrature with $n = 5, 50, 100$, plot the weights vs the abscissas, i.e., the c_k's vs the x_k's that appear in Eq. (7.102).
18. As promised at the start of section 7.5.2, we now discuss another way of showing that the nodal abscissas in Gauss–Legendre quadrature are given by roots of Legendre polynomials. Take $f(x)$ to be a polynomial of degree up to $2n - 1$; divide $f(x)$ with $P_n(x)$, the Legendre polynomial of degree n:

$$f(x) = Q_{n-1}(x)P_n(x) + R_{n-1}(x) \tag{7.243}$$

where the quotient and remainder are polynomials of degree up to $n - 1$. Integrate both sides from -1 to $+1$ and justify why one of these integrals vanishes. Express the non-vanishing integral as a sum over weights and function values, as in our defining Eq. (7.7). Now take the x_k's to be the roots of $P_n(x)$ and re-express the sum using Eq. (7.243) evaluated at the x_k's.
19. Yet another approach to Gaussian quadrature was mentioned at the start of section 7.5.2: the Golub–Welsch algorithm. In a problem in chapter 5 we saw how to cast computing the roots of a Legendre polynomial in the form of evaluating the eigenvalues of tridiagonal Jacobi matrices; we now see how to get the weights from the eigenvectors. Use the matrix $\mathbf{J}$ to produce the corresponding symmetric Jacobi matrix $\tilde{\mathbf{J}}$:

$$\tilde{J}_{i,i} = J_{i,i}, \qquad \tilde{J}_{i-1,i} = \tilde{J}_{i,i-1} = \sqrt{J_{i-1,i}J_{i,i-1}} \tag{7.244}$$

First check (programmatically) that this new matrix $\tilde{\mathbf{J}}$ is similar to $\mathbf{J}$, i.e., it has the same

eigenvalues. Then, find $\tilde{\mathbf{J}}$'s normalized eigenvectors, $\mathbf{v}_j$, and compute the weights from the formula $c_j = 2[(\mathbf{v}_j)_0]^2$. Here $(\mathbf{v}_j)_0$ is the first component of the eigenvector $\mathbf{v}_j$.

20. In Code 3.3, i.e., `psis.py`, we computed the values of Hermite polynomials and their derivatives; then, in a problem in chapter 5 we saw how to compute their roots, by modifying Code 5.4, i.e., `legroots.py`. Following the approach outlined in section 7.5.3 one arrives at the following formula for the weights:

$$c_j = \frac{2^{n+1} n! \sqrt{\pi}}{[H_n'(x_j)]^2} \tag{7.245}$$

which is the Gauss–Hermite analogue of Eq. (7.117). Implement a general routine for Gauss–Hermite quadrature; feel free to use `numpy.sort()` and `numpy.argsort()`. The factorials in our equation for the c_j's can cause problems, but our root-solver doesn't work for too large values of n, anyway, so don't go above $n = 20$ or so. Compare the results with what you get from `numpy.polynomial.hermite.hermgauss()`.

21. Carry out the change of variables $u = \sqrt{x}$ for the integral:

$$I = \int_0^\infty \frac{x^2 + x}{\sqrt{x}} e^{-x} dx \tag{7.246}$$

and observe that you can compute it using Gauss–Hermite quadrature, if you also notice that the integrand is now even. If you solved the previous problem use the function you developed there, otherwise call `numpy.polynomial.hermite.hermgauss()`.

22. Compare the trapezoid rule, Simpson's rule, and Gauss–Legendre quadrature for:

$$I_1 = \int_0^{2\pi} e^{\sin(2x)} dx, \qquad I_2 = \int_0^{2\pi} \frac{1}{2 + \cos x} dx, \qquad I_3 = \int_{-1}^{1} e^{-x^2/2} dx \tag{7.247}$$

Do you understand why the trapezoid rule is not giving terrible results?

23. Apply the trapezoid rule to the following integral:

$$I = \int_0^\pi \sqrt{x} \cos x dx \tag{7.248}$$

(a) Print out a table of values for increasing values of n, checking to see which digits are no longer changing. Do you understand why this function is so slow to converge?

(b) Carry out the change of variables $u = \sqrt{x}$ and then apply the trapezoid rule again. Compare this convergence behavior with that in the previous part of this problem.

24. You will now manipulate the integral I_G from the main text in two distinct ways. First, apply the slicing and then a different transformation for each integral as discussed in Eq. (7.153) and in the following discussion. Second, transform Eq. (7.155) twice, first by taking $v = 1 - u$ and then $w = \sqrt{v}$.

25. Manipulate the integral:

$$I = \int_{10^{-6}}^1 \frac{e^{-x}}{x} dx \tag{7.249}$$

using the techniques of section 7.6.2. Specifically, subtract off the singularity and then use a Taylor expansion where necessary.

26. Evaluate the following four integrals that arise in the study of the *simple pendulum* beyond the small-angle approximation:

$$I_1 = \int_0^{\theta_0} \frac{1}{\sqrt{\cos\theta - \cos\theta_0}}\, d\theta, \qquad I_2 = \int_0^{\pi/2} \frac{1}{\sqrt{1 - A^2 \sin^2\phi}}\, d\phi$$
$$I_3 = \int_0^{\theta_0} \sqrt{\cos\theta - \cos\theta_0}\, d\theta, \qquad I_4 = \int_0^{\pi/2} \frac{1 - \sin^2\phi}{\sqrt{1 - A^2 \sin^2\phi}}\, d\phi \tag{7.250}$$

where $A = \sin(\theta_0/2)$ and we take $\theta_0 = \pi/3$. Of these, I_3 and I_4 appear when you use "action-angle coordinates", whereas I_2 and I_4 are known as *elliptic integrals* of the first and second kind, respectively. Compute these integrals using the adaptive trapezoid rule and Gauss–Legendre quadrature for a comparable number of points.

27. Reproduce both panels of Fig. 7.5. Then, make a pairwise plot using `random.random()`.

28. We now introduce the *equidistribution test* for random-number generators. Split the region from 0 to 1 into, say, $n_b = 10$ bins of equal size. Generate $n = 10^5$ samples in total and count the populations in each bin, n_j, where $j = 1, 2, \ldots, n_b$. We expect to find n/n_b samples in each bin (so 10^4 for our example). Compute the following metric:

$$\chi^2 = \frac{n_b}{n} \sum_{j=1}^{n_b} \left(n_j - \frac{n}{n_b} \right)^2 \tag{7.251}$$

for the three generators used in the previous problem. A "good" χ^2 is equal to the number of degrees of freedom (as you learned in chapter 6). Since $\sum_j n_j = n$, the n_b populations are not fully independent, so we actually have $n_b - 1$ degrees of freedom.

29. The *Chebyshev inequality* for a general random variable X, whose population mean is $\langle X \rangle$, can be expressed as:

$$P\left(|X - \langle X \rangle| \geq \epsilon\right) \leq \frac{\mathrm{var}(X)}{\epsilon^2} \tag{7.252}$$

where ϵ is positive and arbitrarily small. In words, this says that the probability of the random variable's value being more than ϵ away from the population mean is less than the population variance divided with ϵ^2. To show this, apply the definition of the left-hand side, then multiply the integrands by $(x - \langle X \rangle)^2/\epsilon^2$ (why are you allowed to do this?), extend the integration interval from $-\infty$ to $+\infty$, and then employ the definition of the population variance from Eq. (7.177).

Having shown Eq. (7.252), one can apply it to our sample mean $\overline{f}$ which is, after all, a random variable. Then, re-express the right-hand side using Eq. (7.181) and take the limit $n \to \infty$; since we're working with a bounded var$[f(X)]$, the right-hand side goes to 0. Of course, this also means that:

$$\lim_{n\to\infty} P\left(|\overline{f} - \langle f(X) \rangle| < \epsilon \right) = 1 \tag{7.253}$$

This result is known as the *weak law of large numbers*.

30. Given the definition of the estimator e_{var} in Eq. (7.183), show that its expectation is as per Eq. (7.184). To show this, explicitly take the expectation as defined in Eq. (7.175),

following steps similar to those in Eq. (7.180). Crucially, when taking the expectation of $\left[\sum_i f(X_i)/n\right]^2$ you have to distinguish between "diagonal" and "off-diagonal" terms and also use the independence of different samples.

31. Assume that the random variables X_0, X_1, X_2, …, X_{n-1} are independent, identically distributed according to $p(x)$: $p(x)$ could be any distribution with finite variance (i.e., *not* necessarily Gaussian); take each variable to have population mean μ and population variance σ^2.[61] The *central limit theorem* tells us that the sample mean $\overline{x}$ obeys a Gaussian distribution (asymptotically).

    To see this, we first define the *characteristic function* of a random variable Z to be $\varphi_Z(t) = \langle e^{itZ} \rangle$. Write down the characteristic function of the variable $Z = \sum_i Y_i/\sqrt{n}$ and show that it is equal to the characteristic function of one of the Y_i's, raised to the n-th power. Then, write down the characteristic function of the variable $Y_0 = (X_0 - \mu)/\sigma^2$ and Taylor expand it. Combine the last two results to show that $\varphi_Z(t) \to e^{-t^2/2}$ as $n \to \infty$. This is the characteristic function of a normal distribution with zero mean and variance of one; use this result to show that the sample mean $\overline{x}$ obeys a normal distribution with mean μ and variance σ^2/n.

32. Produce a plot that looks like the right panel of Fig. 7.7, for the integral of Eq. (7.198), i.e., $I = \int_0^1 e^{-x} \cos x dx$, for uniformly and exponentially distributed numbers.

33. We now discuss the *Box–Muller method* of producing normal deviates; this makes a detour through two-dimensional distributions. Recall that in the one-dimensional case we carried out the integration in Eq. (7.195); the analogue here is the joint cumulative distribution function:

$$u(x_0, x_1) = \int_{-\infty}^{x_0} \int_{-\infty}^{x_1} w(x_0', x_1') dx_0' dx_1' = \int_{-\infty}^{x_0} \int_{-\infty}^{x_1} \frac{e^{-(x_0')^2/2}}{\sqrt{2\pi}} \frac{e^{-(x_1')^2/2}}{\sqrt{2\pi}} dx_0' dx_1' \qquad (7.254)$$

    As an earlier problem showed, the Gaussian integral that seems complicated in one dimension becomes near-trivial in two dimensions (in polar coordinates). You'll find that your result for $u(r, \theta)$ depends on r and θ separately, i.e., it is of the form $u_0(r)u_1(\theta)$. Inverting these two functions, you can get $r(u_0)$ and $\theta(u_1)$. Going back to Cartesian coordinates (while recalling that if u_0 is a uniform deviate, then so is $1 - u_0$), show that:

$$x_0 = \sqrt{-2 \log u_0}\, \cos(2\pi u_1), \qquad x_1 = \sqrt{-2 \log u_0}\, \sin(2\pi u_1) \qquad (7.255)$$

    This accomplishes our task of producing two normal deviates (x_0 and x_1) from two uniform deviates (u_0 and u_1).

34. Generalize Eq. (7.162) so that you can tackle the following five-dimensional integral:

$$I = \int_{-1}^{1} du\, dv\, dw\, dx\, dy \left(\frac{2 + u + v}{5 + w + x} \right)^y \qquad (7.256)$$

    using Gauss–Legendre quadrature and $\mathcal{N} = 100$–200. Then, apply uniform multidimensional Monte Carlo integration as per Eq. (7.211), with $\mathcal{N} = 10^7$–10^8. Observe that the $\mathcal{N}$'s in the two cases are vastly different, but the total runtimes not so much.

35. This problem shows that the $w(\mathbf{X})$ distribution is stationary when detailed balance holds, employing a different argument than in the main text.

[61] Using our earlier notation, these would be $\langle X_i \rangle$ and $\mathrm{var}(X_i)$, which are the same for different values of i.

(a) Re-express Eq. (7.214) so that it takes the form:

$$p_i(\mathbf{X}) = \int p_{i-1}(\mathbf{Y}) T(\mathbf{Y} \to \mathbf{X}) d^d Y \tag{7.257}$$

Feel free to employ the normalization of the transition probability.

(b) Integrate Eq. (7.213) over $\mathbf{Y}$, use the normalization again, and combine your result with Eq. (7.257) to show that $w(\mathbf{X})$ is a stationary distribution.

36. The one-dimensional integral $\int_0^\infty e^{-x} x^2 dx$ is of the form $\int_0^\infty w(x) f(x) dx$, for $w(x) = e^{-x}$ and $f(x) = x^2$. Implement the Metropolis algorithm for $n = 10^8$; observing that you can also employ Gauss–Laguerre quadrature, re-compute this integral using the function `numpy.polynomial.laguerre.laggauss()` and $n = 2$.

37. We now show that, when the variational energy E_V for the trial wave function ψ_T gives the ground-state energy E_0, then ψ_T *is* the ground-state wave function; imagine slightly perturbing ψ_T such that Eq. (7.232) turns into:

$$E_0 + \Delta E = \frac{(\langle \psi_T| + \langle \Delta\psi_T|) \hat{H} (|\psi_T\rangle + |\Delta\psi_T\rangle)}{(\langle \psi_T| + \langle \Delta\psi_T|) (|\psi_T\rangle + |\Delta\psi_T\rangle)} \tag{7.258}$$

Manipulate this equation by neglecting terms of order $\langle \Delta\psi_T | \Delta\psi_T \rangle$. Do this a second time in order to move all $|\Delta\psi_T\rangle$ terms onto the numerator. Then, set $\Delta E = 0$ to show that $\langle \Delta\psi_T | (\hat{H} - E_0) | \psi_T \rangle = 0$. Discuss what this equation implies on the status of $|\psi_T\rangle$.

38. This problem studies the *autocorrelation function*, defined to be:[62]

$$C(k) = \frac{\overline{f f_k} - \overline{f}^2}{\overline{f^2} - \overline{f}^2} \tag{7.259}$$

Here we repeatedly use the arithmetic average, defined in Eq. (7.179). It involves terms we've encountered before, like $\overline{f^2}$ and $\overline{f}^2$ which are used in Eq. (7.183), as well as a new, strange-looking term, $\overline{f f_k}$; this is simply shorthand for:

$$\overline{f f_k} = \frac{1}{\mathcal{N} - k} \sum_{i=0}^{\mathcal{N}-k-1} f(X_i) f(X_{i+k}) \tag{7.260}$$

In this equation, the value of f at a given sample is multiplied with that of f for the sample that is k steps apart (and such pairwise multiplications are carried out *for the entire chain*); for example, $k = 1$ would connect $f(X_i)$ with $f(X_{i+1})$. Of course, given our definition, $C(0) = 1$; $C(1)$ would be smaller, and then at some point $C(k)$ would be small enough that you wouldn't have to worry about it. The value of k at which this happens (the *correlation length*) tells you how many steps you should skip in order to minimize the correlation between succeeding walker configurations.

For our random walk in `vmc.py`, E_L will play the role of f in the definition above. Note that you need to take consecutive measurements to apply Eq. (7.259), i.e., $n_m = 1$. Your task is to plot the autocorrelation function for $k = 0, 1, \ldots, 100$ for $\mathcal{N} = 10^6$.

[62] This is similar to concepts we introduced in chapter 6: in the present context of a random walk the numerator in Eq. (7.259) is known as the *autocovariance*. In principle one should generalize our expression for the variance in the main text to also involve the autocovariance: in the earlier problem on $\langle e_{var} \rangle$ this term vanished, because the samples were independent there.

39. In our discussion of the Metropolis–Hastings algorithm we provided a general formula for the acceptance probability, Eq. (7.219). This, however, is not unique; an alternative choice, known as *Barker's algorithm*, is:

$$\alpha(\mathbf{X} \to \mathbf{Y}) = \frac{R(\mathbf{X} \to \mathbf{Y})}{1 + R(\mathbf{X} \to \mathbf{Y})} \tag{7.261}$$

First, show that this acceptance probability allows you to satisfy detailed balance. Second, update `vmc.py` to use this new criterion.

40. Rewrite `vmc.py`, breaking down the contributions to E_V in terms of kinetic, one-body, and two-body energies. Which contribution is the most important one?
41. Rewrite the function `ekin()` in `eloc.py` such that you don't use any finite-difference approximation (and therefore also no step size h). This means you first have to analytically take the second derivatives and then code up your expressions; make sure that you (analytically) cancel any terms that do cancel. Compare your output with that of `eloc.py` for different values of h.
42. To make this problem more manageable, study $n_p = 4$ and $n_d = 2$. Observe from the first equality in Eq. (7.236) that we have to compute eight-dimensional integrals. However, if you write out $\hat{H}_1$ you will realize that the sum contains six terms, each of which only couples a single pair of particles (the j-th and k-th particle each time). Thus, the eight-dimensional integral decouples into an "easy" four-dimensional one and a "hard" four-dimensional one. Carrying these out by Gauss–Legendre (or using a software package), you should compare with the output of `vmc.py`; if it makes the comparison easier, choose smaller values of g.
43. Generalize `vmc.py` to handle particles that obey Fermi–Dirac statistics (do you expect the energy to increase or decrease?). The many-particle wave function is not a simple product of individual ϕ's as in Eq. (7.226), but has to be antisymmetrized, thereby giving rise to a *Slater determinant* involving the ϕ's; feel free to use Eq. (4.150) and `ludec.py`. For concreteness, study only the case of four particles in three dimensions. Since you're dealing with fermions, you can place only a single particle in the state $n_x = n_y = n_z = 0$; the next particle goes in $n_x = 1, n_y = n_z = 0$, and so on. When you code these up you can use `hermite()` from `psis.py`, or simply hard-code the fact that $H_1(x) = 2x$.
44. We will now come up with a better trial wave function ψ_T than the one we used in `vmc.py`, by employing different variational parameters in each direction, i.e., α_x, α_y, α_z. For $\mathcal{N} = 10^3$ and only four or five values of α in each direction (in order to speed the computations up) set up a three-dimensional grid for the three parameters α_x, α_y, α_z and see if you can find any regions that lead to smaller values of E_V than what we found in Fig. 7.8.