Derivatives 3

I'll teach you differences.

William Shakespeare

3.1 Motivation

In this chapter and in all the following ones we start out with a section titled "Motivation", which has a dual purpose: (a) to provide examples of the chapter's theme drawn from the study of physics, and (b) to give a mathematical statement of the problem(s) that will be tackled in later sections.

3.1.1 Examples from Physics

Here are three examples of the use of derivatives in physics:

1. In classical mechanics, the definition of the *velocity* of a single particle is:

$$\mathbf{v} = \frac{d\mathbf{r}}{dt} \tag{3.1}$$

   where $\mathbf{r}$ is the position of the particle in a given reference frame. This definition tells us that the velocity is the time derivative of the position.
2. In classical electromagnetism, the equation connecting the electric field $\mathbf{E}$ with the vector potential $\mathbf{A}$ and scalar potential ϕ is:

$$\mathbf{E} = -\nabla\phi - \frac{\partial \mathbf{A}}{\partial t} \tag{3.2}$$

   The right-hand side contains both spatial derivatives and a time derivative.
3. The Lagrangian density for the vibrations of a continuous rod can be expressed as:

$$\mathcal{L} = \frac{1}{2}\left[\frac{1}{c^2}\left(\frac{\partial\varphi}{\partial t}\right)^2 - \left(\frac{\partial\varphi}{\partial x}\right)^2\right] \tag{3.3}$$

   Here c is the velocity of longitudinal elastic waves. The dynamical variable is $\varphi(x, t)$. As you can see, the spatial and temporal derivatives of this quantity together make up the Lagrangian density. Something very similar to this expression appears in state-of-the-art lattice field theory computations.

3.1.2 The Problem to Be Solved

More generally, our task is to evaluate the derivative of $f(x)$ at a specific point, $f'(x)$. If we have an analytical expression for the x-dependence of the function $f(x)$, then this problem is in principle trivial , even though humans are error-prone when dealing with complicated expressions. However, in many cases we, instead, have a set of n discrete data points (i.e., a table) of the form $(x_i, f(x_i))$ for $i = 0, 1, \ldots, n-1$.[1] This often happens when we are dealing with a computationally demanding task: while we could produce more-and-more sets of points, in practice, we are limited by the time it takes to produce those.

One could approach this problem in a number of ways. First, we can use interpolation or data fitting (see chapter 6) to produce a new function that approximates the data reasonably well, and then apply *analytical differentiation* to that function. This is especially helpful when we are dealing with noisy data. On the other hand, just because a function approximates a set of data points quite well, doesn't mean that it also captures information on the derivative (or even higher-order derivatives) of the function. In other words, interpolating/fitting and then taking the derivative doesn't provide much guidance regarding the error involved when we're interested in the derivative (do we know we're using the right analytical form?). The second class of approach is helpful in systematizing our ignorance: it is called the *finite-difference* approach, also known simply as numerical differentiation. In a nutshell, it makes use of the Taylor series expansion of the function we are interested in differentiating, around the specific point where we wish to evaluate the derivative. The third class of approach is known as *automatic differentiation*: this is as accurate as analytical differentiation, but it deals with numbers instead of mathematical expressions.

3.2 Analytical Differentiation

Derivatives like those discussed in the previous section are defined in the usual way:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{3.4}$$

This gives us the derivative of a function f at the point x in terms of a limit. In practice, we typically don't apply this definition, but use, instead, standard differentiation rules for powers, quotients, products, as well as the well-known expressions for the derivative of several special functions. As you learned a long time ago, the application of these rules can help you differentiate any function you wish to, e.g.:

$$\frac{d}{dx} e^{\sin(2x)} = 2\cos(2x)\, e^{\sin(2x)} \tag{3.5}$$

You could similarly evaluate the second derivative, the third derivative, and so on.

[1] Note that when labelling our n points we start at 0 and end at $n-1$: this is consistent with Python's 0-indexing.

In most of this book, we are interested in "computing", namely plugging actual numbers into mathematical expressions. Even so, there are times when carrying out a numerical evaluation might obscure underlying simplicities. Other times, carrying analytical calculations out by hand can get quite tedious. The solution for these scenarios is to use a computer algebra package. Such packages allow one to carry out symbolic manipulations on the computer. `Sage` is a mathematical software system with a "Python-like" syntax that was intended to be an alternative to solutions like `Maple`, `Mathematica`, or `Matlab`. In one of the problems, we focus on a much more lightweight solution, namely `SymPy` (actually included as part of `Sage`): this is a Python module (that can be used like any other Python module) to carry out symbolic manipulations using Python idioms whenever possible.

3.3 Finite Differences

Let us return to the definition of the derivative given in Eq. (3.4):

$$\left.\frac{df(x)}{dx}\right|_{\tilde{x}} = \lim_{h\to 0} \frac{f(\tilde{x}+h) - f(\tilde{x})}{h} \tag{3.6}$$

where we are slightly changing our notation to show that this definition allows us to evaluate the derivative of $f(x)$ at the point $\tilde{x}$.[2] The obvious thing that comes to mind is to try to use this formula but, instead of taking the limit $h \to 0$, simply take h to be "small". There are many questions that immediately emerge with this *ad hoc* approach: (a) do we have any understanding of the errors involved?, (b) do we know what "small" means?, (c) do we realize that, typically, the smaller h becomes, the smaller the numerator becomes? Focusing on the last point: as we're making h smaller and smaller, we're producing a smaller and smaller numerator $f(\tilde{x}+h) - f(\tilde{x})$, since we're evaluating the function f at two points that are just next to each other: as if that wasn't enough, then we're simply dividing with that tiny number h (since it's in the denominator), magnifying any mistakes we made in the evaluation of the numerator.

In the rest of this section, we will try to remedy the problems of this *ad hoc* approach, in order to be more systematic. As mentioned above, we will make repeated use of the Taylor series expansion of the function we are interested in differentiating.

3.3.1 Noncentral-Difference Approximations

Forward Difference

With the aforementioned disclaimer that we don't distinguish between x and $\tilde{x}$, let us start from the all-important Taylor expansion of $f(x+h)$ around x, see Eq. (C.1):

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + \frac{h^4}{24}f^{(4)}(x) + \cdots \tag{3.7}$$

[2] Informally, this distinction between a general point x and a specific point $\tilde{x}$ is often passed over.

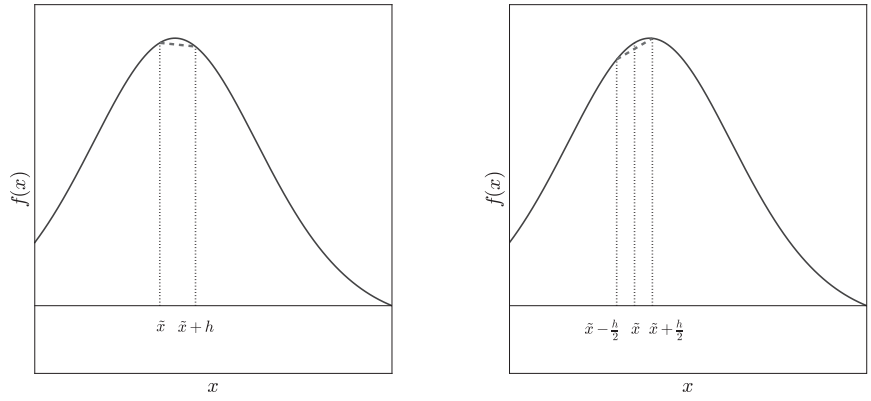


Fig. 3.1 First approximation of a first derivative: forward (left), central (right)

This can be trivially re-arranged to give:

$$f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{h}{2} f''(x) + \cdots \tag{3.8}$$

This naturally leads to an approximation for the value of the derivative of $f(x)$ at x:

$$f'(x) = \frac{f(x+h) - f(x)}{h} + O(h) \tag{3.9}$$

known as the *(first) forward-difference approximation*. To be clear, this approximation consists in using only the fraction on the right-hand side to evaluate the derivative: as the second term on the right-hand side shows, this suffers from an error $O(h)$: this is possibly not too bad when h is small. This recipe is called a *forward* difference because it starts at x and then moves in the forward/positive direction to $x + h$. Graphically, this is represented in the left panel of Fig. 3.1. Clearly, the forward difference is nothing other than the slope of the line segment connecting $f(x)$ and $f(x + h)$. This is not always a great approximation: for the example chosen, we would expect the slope of $f(x)$ to be positive.

Note that the formula we arrived at for the forward difference happens to be identical to the formula that we qualitatively discussed after Eq. (3.6): the difference here is that, due to the derivation starting from the Taylor series, we have a handle on the error this approximation corresponds to. It may still suffer from the issues mentioned above (tiny numerator, tiny denominator), but at least now we have some guidance on how well we're doing: if the h is not too small (so that we're still away from major roundoff issues), halving the h should double the quality of the approximation (in absolute terms).

Incidentally, we used in Eq. (3.9) the O symbol, which you may have not encountered before. This is known as big-O notation. For our purposes, it is simply a concise way of encapsulating only the dependence on the most crucial parameter, without having to worry about constants, prefactors, etc. Thus, $O(h)$ means that the leading error is of order h. Since h will always be taken to be "small", we see that an error $O(h)$ is much larger/worse than, say, an error $O(h^6)$. Though there could, in principle, also be prefactors that complicate

such a comparison, the essential feature is captured by the power-law (or other) dependence attached to the O symbol.[3]

Backward Difference

At this point, we observe that we started our derivation in Eq. (3.7) with $f(x+h)$ and then ended up with the prescription Eq. (3.9) for the forward difference. Obviously, we could just as well have started with the Taylor expansion of $f(x-h)$:

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f'''(x) + \frac{h^4}{24}f^{(4)}(x) + \cdots \tag{3.10}$$

Again, this can be trivially re-arranged to give:

$$f'(x) = \frac{f(x) - f(x-h)}{h} + \frac{h}{2}f''(x) - \cdots \tag{3.11}$$

This naturally leads to an approximation for the value of the derivative of $f(x)$ at x:

$$f'(x) = \frac{f(x) - f(x-h)}{h} + O(h) \tag{3.12}$$

known as the *(first) backward-difference approximation*. This is called a *backward* difference because it starts at x and then moves in the negative direction to $x-h$. It is nothing other than the slope of the line segment connecting $f(x-h)$ and $f(x)$. The forward and backward differences are very similar, so we focus on the former, for the sake of concreteness. These are collectively known as *noncentral differences*, as used in the title of this section. "Noncentral" is employed in contradistinction to "central", which we discuss in the following section. Before we do that, though, we will spend some time discussing the errors of the forward-difference approximation in more detail.

Error Analysis for the Forward Difference

We've already seen above that we want the h to be small[4] but, on the other hand, we don't want to make the h too small, as that will give rise to roundoff errors. Clearly, the best-possible choice of h will be somewhere in the middle.

We're here dealing with the combination of an approximation error $\mathcal{E}_{app}$, coming from the truncation of the Taylor series, and a roundoff error $\mathcal{E}_{ro}$, coming from the subtraction and division involved in the definition of the forward difference. In chapter 2 we defined the absolute error as "approximate minus exact", see Eq. (2.5); here we're using a new symbol, $\mathcal{E}$, to denote the magnitude of the absolute error. The magnitude of the approximation error

[3] Also, note that here we are interested in small h, in which case $O(h^2)$ is "better" than $O(h)$. In later chapters, we will be examining the dependence of various quantities on the required number of function evaluations n; in that case $O(n^2)$ will be "worse" than $O(n)$.

[4] So that the term $O(h)$ we're chopping off doesn't matter very much.

is immediately obvious, given Eq. (3.8):

$$\mathcal{E}_{app} = \frac{h}{2}|f''(x)| \tag{3.13}$$

Actually this result is not totally trivial: from what we know about the Lagrange remainder in a Taylor series, this expression should involve $|f''(\xi)|$, where ξ is a point between x and $x + h$. However, since h is small, it is a reasonable approximation to take $|f''(\xi)| \approx |f''(x)|$; we will make a similar approximation (more than once) below.

Turning to $\mathcal{E}_{ro}$, we remember that we are interested in evaluating $(f(x+h) - f(x))/h$. We focus on the numerator: this is subtracting two numbers that are very close to each other, bringing to mind the discussion in section 2.2.2. As per Eq. (2.20), to find the absolute error of the subtraction $f(x + h) - f(x)$ we add together the absolute errors in $f(x + h)$ and in $f(x)$. As per Eq. (2.51), we see that the absolute error in $f(x + h) - f(x)$ is $f(x)2\epsilon_m$, where we assumed $f(x+h) \approx f(x)$ and that the relative error for each function evaluation is approximated by the machine error. If we now ignore the error incurred by the division by h (which we know is a reasonable thing to do), we see that the absolute error in evaluating the forward difference, $(f(x + h) - f(x))/h$, will simply be the absolute error in the numerator divided by h:

$$\mathcal{E}_{ro} = \frac{2|f(x)|\epsilon_m}{h} \tag{3.14}$$

We can now explicitly see that for the case under study the approximation error decreases as h gets smaller, whereas the roundoff error increases as h gets smaller. Adding the approximation error together with the roundoff error gives us:

$$\mathcal{E} = \mathcal{E}_{app} + \mathcal{E}_{ro} = \frac{h}{2}|f''(x)| + \frac{2|f(x)|\epsilon_m}{h} \tag{3.15}$$

To minimize this total error, we set the derivative with respect to h equal to 0:

$$\frac{1}{2}|f''(x)| - \frac{2|f(x)|\epsilon_m}{h_{opt}^2} = 0 \tag{3.16}$$

where we called the h value that minimizes the total absolute error h_{opt}. For future reference, we note that Eq. (3.16) can be manipulated to give:

$$\frac{h_{opt}}{2}|f''(x)| = \frac{2|f(x)|\epsilon_m}{h_{opt}} \tag{3.17}$$

Our equation in Eq. (3.16) can be solved for the optimal value of h giving us:

$$h_{opt} = \sqrt{4\epsilon_m \left|\frac{f(x)}{f''(x)}\right|} \tag{3.18}$$

These results can be plugged into Eq. (3.15) to give the smallest possible error in the forward difference. We do this in two steps. First, we use Eq. (3.17) to find:

$$\mathcal{E}_{opt} = h_{opt}|f''(x)| \tag{3.19}$$

Second, we plug Eq. (3.18) into our latest result to find:

$$\mathcal{E}_{opt} = \sqrt{4\epsilon_m |f(x)f''(x)|} \tag{3.20}$$

For concreteness, assume that $f(x)$ and $f''(x)$ are of order 1. Our result in Eq. (3.18) then tells us that we should pick $h_{opt} = \sqrt{4\epsilon_m} \approx 3 \times 10^{-8}$. In that case, the (optimal/minimum) absolute error is also $\mathcal{E}_{opt} = \sqrt{4\epsilon_m} \approx 3 \times 10^{-8}$. These numbers might, of course, look quite different if the value of the function (or its second derivative) is much greater than or less than 1. Note that an error of 10^{-8} is not that impressive: in section 3.2 on analytical differentiation we had *no* differentiation error: the only error involved was the function evaluation, which is typically a significantly less important problem.

Keep in mind that statements like "the error in this approach is $O(h)$" (referring to the approximation error) are only true for reasonably well-behaved functions. If the corresponding derivative is infinite or doesn't exist, then you cannot rely on the straightforward results on scaling we discuss here.

3.3.2 Central-Difference Approximation

We will now try to find a way to approximate the derivative using a finite difference, but more accurately than with the forward/backward difference. As before, we start from the Taylor expansion of our function around x, but this time we choose to make a step of size $h/2$ (as opposed to the h in the previous section):

$$f\left(x + \frac{h}{2}\right) = f(x) + \frac{h}{2}f'(x) + \frac{h^2}{8}f''(x) + \frac{h^3}{48}f'''(x) + \frac{h^4}{384}f^{(4)}(x) + \cdots \tag{3.21}$$

We can also write down a similar Taylor expansion for a move in the negative direction:

$$f\left(x - \frac{h}{2}\right) = f(x) - \frac{h}{2}f'(x) + \frac{h^2}{8}f''(x) - \frac{h^3}{48}f'''(x) + \frac{h^4}{384}f^{(4)}(x) - \cdots \tag{3.22}$$

Comparing the last two equations, we immediately see that adding them or subtracting them can lead to useful patterns: the sum contains only even derivatives whereas the difference contains only odd derivatives. In the present case, we subtract the second equation from the first: all even derivatives along with the $f(x)$ term cancel. We can now re-arrange the result, solving for $f'(x)$:

$$f'(x) = \frac{f\left(x + \frac{h}{2}\right) - f\left(x - \frac{h}{2}\right)}{h} - \frac{h^2}{24}f'''(x) + \cdots \tag{3.23}$$

This naturally leads to an approximation for the value of the derivative of $f(x)$ at x:

$$f'(x) = \frac{f\left(x + \frac{h}{2}\right) - f\left(x - \frac{h}{2}\right)}{h} + O(h^2) \tag{3.24}$$

known as the *(first) central difference approximation*. As the second term on the right-hand side shows, this suffers from an error $O(h^2)$: since h is generally small, h^2 is even smaller.

Just like for the forward-difference approximation in Eq. (3.9), computing the central difference requires only two function evaluations. This is called the *central* difference because these two evaluations are at $x - h/2$ and at $x + h/2$, i.e., they are centered at x: just like for the forward difference, these two points are h apart. Graphically, this is represented in the right panel of Fig. 3.1. Clearly, the central difference is nothing other than the slope of the line segment connecting $f(x - h/2)$ and $f(x + h/2)$. As the figure shows, for the example chosen this is a much better approximation than the forward difference in the left panel. We will discuss this approximation's error budget below: for now, note that if the h is not too small (so that we're still away from major roundoff issues), halving the h should quadruple the quality of the approximation (in absolute terms).

There exists one (very common in practice) situation where a central-difference approximation is simply not usable: if we have a set of n discrete data points (i.e., a table) of the form $(x_i, f(x_i))$ for $i = 0, 1, \ldots, n-1$ we will not be able to use a central difference to approximate the derivative at x_0 or at x_{n-1}: for any of the "middle" points we could always use two evaluations (one on the left, one on the right), but for the two endpoints we simply don't have points available "on the other side", so a forward/backward difference is necessary there.

Error Analysis for the Central Difference

We wish the h to be small,[5] but we should be cautious about not making the h too small, lest that give rise to roundoff errors. Once again, the best-possible choice of h will be somewhere in the middle. As you'll show in a problem, for this case adding the approximation error together with the roundoff error gives us:

$$\mathcal{E} = \mathcal{E}_{app} + \mathcal{E}_{ro} = \frac{h^2}{24}|f'''(x)| + \frac{2|f(x)|\epsilon_m}{h} \tag{3.25}$$

You will also show that minimizing this total error leads to:

$$h_{opt} = \left(24\epsilon_m \left|\frac{f(x)}{f'''(x)}\right|\right)^{1/3}, \quad \mathcal{E}_{opt} = \left(\frac{9}{8}\epsilon_m^2[f(x)]^2|f'''(x)|\right)^{1/3} \tag{3.26}$$

For concreteness, assume that $f(x)$ and $f'''(x)$ are of order 1. Our result in Eq. (3.26) then tells us that we should pick $h_{opt} = (24\epsilon_m)^{1/3} \approx 2\times10^{-5}$. In that case, the (optimal/minimum) absolute error is $\mathcal{E}_{opt} = (9\epsilon_m^2/8)^{1/3} \approx 4\times10^{-11}$. These numbers might, of course, look quite different if the value of the function (or its third derivative) is much greater than or less than 1. This time our error of 10^{-11} is much better than that for the forward difference, though it's still likely worse than what we got in section 3.2 using analytical differentiation.

We just observed that the error for the central difference (10^{-11}) is considerably smaller than that for the forward difference (10^{-8}). Intriguingly, the optimal step size for the central difference (10^{-5}) is orders of magnitude *larger* than the optimal step size for the forward difference (10^{-8}): the better algorithm allows us to "get away with" a larger h.

[5] As before, so that the term $O(h^2)$ we're chopping off doesn't matter very much.

3.3.3 Implementation

We now turn to an example: we will employ the same function, $f(x) = e^{\sin(2x)}$, that we analytically differentiated in section 3.2. For concreteness, we study the derivative at a fixed x, namely $x = 0.5$. We will examine the absolute errors in approximating the derivative using forward and central differences in Code 3.1. We first define one Python function, `f()`, corresponding to the mathematical function whose derivative we're interested in calculating, and then another Python function, `fprime()`, corresponding to the derivative itself: this will be used as an analytical benchmark, with which to compare the difference formula results. We then define functions giving the forward and central difference results, which take in as an argument the function we wish to differentiate (which could be changed later). We then re-encounter the `if __name__ == '__main__':` idiom. This will come in handy in a later section, when we will employ the functions defined in this program without running the rest of the code.

We use a few list comprehensions to store the step sizes and the difference-formula results corresponding to them. In order to make the output more legible than it would be by default, we employ a format string as discussed in chapter 1. The only new feature here is that we store the format string into the variable `rowf`, in order to make the line containing the `print()` itself easier to understand. We print things out this way because we will be faced with a cascade of zeros. The output of running this code is:

```
h     abs. error in fd   abs. error in cd
1e-01 0.3077044583376249 0.0134656094697689
1e-02 0.0260359156901186 0.0001350472492652
1e-03 0.0025550421497806 0.0000013505116288
1e-04 0.0002550180941236 0.0000000135077878
1e-05 0.0000254969542519 0.0000000001051754
1e-06 0.0000025492660578 0.0000000002500959
1e-07 0.0000002564334673 0.0000000011382744
1e-08 0.0000000255070782 0.0000000189018428
1e-09 0.0000000699159992 0.0000000699159992
1e-10 0.0000021505300500 0.0000021505300500
1e-11 0.0000332367747395 0.0000111721462455
```

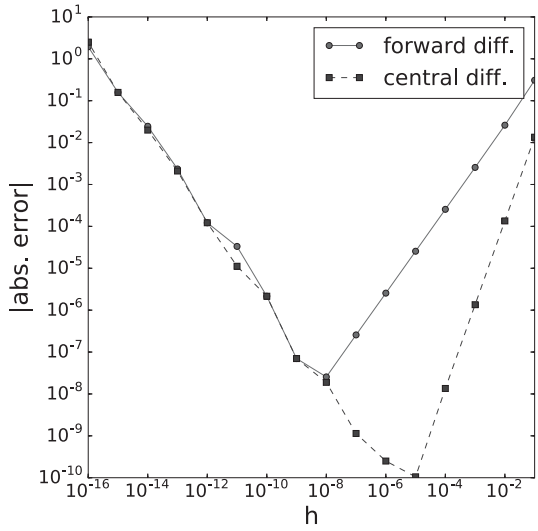
Let's first look at the forward-difference results: we see that as h gets smaller by an order of magnitude, the absolute error also gets smaller by an order of magnitude: recall that for this approach the approximation error is $O(h)$. The minimum absolute error turns out to be $\approx 2.5 \times 10^{-8}$, which appears for the step size $h = 10^{-8}$. This is qualitatively consistent with what we saw in the section on the error analysis above; this time around we don't have guarantees that the function (or derivative) values are actually 1. Actually, the absolute error we found here is slightly smaller than expected, due to a cancellation of errors that we couldn't generally assume was going to be present (e.g., for another value of x). Beyond this point, as we keep reducing h, we see that the absolute error starts increasing: the roundoff error is now dominating, leading to increasingly poor results.

Code 3.1 finitediff.py

```
from math import exp, sin, cos

def f(x):
     return exp(sin(2*x))

def fprime(x):
     return 2*exp(sin(2*x))*cos(2*x)

def calc_fd(f,x,h):
     fd = (f(x+h) - f(x))/h
     return fd

def calc_cd(f,x,h):
     cd = (f(x+h/2) - f(x-h/2))/h
     return cd

if __name__ == '__main__':
     x = 0.5
     an = fprime(x)

     hs = [10**(-i) for i in range(1,12)]
     fds = [abs(calc_fd(f,x,h) - an) for h in hs]
     cds = [abs(calc_cd(f,x,h) - an) for h in hs]

     rowf = "{0:1.0e} {1:1.16f} {2:1.16f}"
     print("h     abs. error in fd     abs. error in cd")
     for h,fd,cd in zip(hs,fds,cds):
          print(rowf.format(h,fd,cd))
```

The results for the central-difference formula are completely analogous: we see that as h gets smaller by an order of magnitude, the absolute error also gets smaller by two orders of magnitude: recall that for this approach the approximation error is $O(h^2)$. This process reaches a minimum at $h = 10^{-5}$: the absolute error there is $\approx 10^{-10}$, similarly to what was seen in our earlier discussion of the error budget. As before, reducing the h further gives disappointing results, since the roundoff error dominates beyond that point.

After you solve the corresponding problem, you will produce Fig. 3.2. The step size h gets smaller as we go to the left, whereas the absolute value of the absolute error gets smaller as we go to the bottom of the plot. The behavior encountered in this plot (for a



Log-log plot using forward-difference and central-difference formulas **Fig. 3.2**

specific function at a specific point) is pretty generic. On the right part of the plot we are dominated by the truncation/approximation error: there, the central difference is clearly superior, as the absolute error is consistently smaller. In this region, we can easily see our earlier statements in action: as h changes by an order of magnitude, the error of the forward difference changes by an order of magnitude, whereas the error of the central difference changes by two orders of magnitude. As we keep moving to the left, beyond a certain point roundoff error starts dominating and the forward difference is as good (or as bad) as the central difference. We stop plotting when the error is so large that we are completely failing at evaluating the derivative.

It's worth pointing out that the source of all the roundoff problems that *all* finite-difference formulas suffer from is that in each case *the sum of all the function evaluation coefficients is zero!* This is pretty obvious at this point, since we've only seen two finite-difference formulas (Eq. (3.9) which contained $+f(x + h) - f(x)$ and Eq. (3.24) which contained $+f(x + h/2) - f(x - h/2)$), but you should keep it in mind as you proceed. Having terms that nearly cancel each other invites trouble.

3.3.4 More Accurate Finite Differences

Up to this point, we've seen noncentral and central finite differences; the forward difference had an error $O(h)$ and the central difference an error $O(h^2)$. It's important to note that both the forward difference, Eq. (3.9), and the central difference, Eq. (3.24), require two function evaluations to calculate a finite-difference ratio.

As you will show in the problem set, one can produce the following approximation for the value of the derivative of $f(x)$ at x:

$$f'(x) = \frac{4f\left(x+\frac{h}{2}\right) - f(x+h) - 3f(x)}{h} + \frac{h^2}{12}f'''(x) + \cdots \tag{3.27}$$

known as the *second forward-difference approximation*. It has this name because: (a) it also requires evaluations of the function at x and other points to the right (so it is a forward difference), and (b) while being a forward difference, it suffers from an error $O(h^2)$, i.e., it is more accurate than the forward-difference approximation in Eq. (3.9), which has an error of $O(h)$ (so Eq. (3.27) is the *second* forward-difference approximation). While this new approximation seems to be as good as the central difference in Eq. (3.24) which also has an error $O(h^2)$, it accomplishes this at the cost of requiring three function evaluations: $f(x)$, $f(x + h/2)$, and $f(x + h)$.

In the problem set you will also produce yet another approximation for the value of the derivative of $f(x)$ at x:

$$f'(x) = \frac{27f\left(x+\frac{h}{2}\right) + f\left(x-\frac{3}{2}h\right) - 27f\left(x-\frac{h}{2}\right) - f\left(x+\frac{3}{2}h\right)}{24h} + \frac{3}{640}h^4 f^{(5)}(x) + \cdots \tag{3.28}$$

known as the *second central-difference approximation*, for obvious reasons. This approximation has an error $O(h^4)$, making it the most accurate finite-difference formula (for the first derivative) that we've encountered. However, it accomplishes this at the cost of requiring four function evaluations: $f(x + 3h/2)$, $f(x + h/2)$, $f(x - h/2)$, and $f(x - 3h/2)$. It's worth emphasizing what we noted above: the sum of all the function evaluation coefficients is zero for this formula, too ($27 + 1 - 27 - 1 = 0$.)

It should be clear by now that one can keep including more points, taking Taylor expansions, multiplying and adding/subtracting those together, to arrive at formulas (whether central or noncentral) that are of even higher order in accuracy. As we've already noted, the problem with that approach is that one needs increasingly many function evaluations: in practice, evaluating $f(x)$ at different points is a costly task, so it turns out that more accurate expressions like those we've seen in this section are not too commonly encountered "in the wild". (You should also try to come up with a $O(h^4)$ method by using $f(x+h/2)-f(x-h/2)$ together with $f(x + h) - f(x - h)$.)

3.3.5 Second Derivative

In practice, we also need higher-order derivatives: the second derivative is incredibly important in all areas of physics. It should come as no surprise that one can set up forward, backward, and central difference expressions (of increasing sophistication) that approximate the second derivative. It is common to derive the simplest possible finite-difference formula for the second derivative by saying that the second derivative is the first derivative

of the first derivative, which symbolically translates to:

$$f''(x) \approx \frac{f'\left(x+\frac{h}{2}\right) - f'\left(x-\frac{h}{2}\right)}{h} \tag{3.29}$$

This is nothing other than the central-difference formula, Eq. (3.24), applied once. Applying it twice more (on the right-hand side) would give us an explicit approximation formula for the second derivative in terms of function evaluations only.

Instead of pursuing that avenue, however, we will here follow the spirit of the previous sections, where we repeatedly used Taylor expansions, manipulating them to eliminate terms of our choosing. Specifically, we look, once again, at Eq. (3.21) and Eq. (3.22). At this point, we note that subtracting one of these equations from the other leads to odd derivatives alone. It is just as easy to see that summing these two Taylor expansions together:

$$f\left(x+\frac{h}{2}\right) + f\left(x-\frac{h}{2}\right) = 2f(x) + \frac{h^2}{4}f''(x) + \frac{h^4}{192}f^{(4)}(x) + \cdots \tag{3.30}$$

leads to even derivatives alone. Since we're interested in approximating the second derivative, we realize we're on the right track. Our result can be trivially re-arranged to give:

$$f''(x) = 4\frac{f\left(x+\frac{h}{2}\right) + f\left(x-\frac{h}{2}\right) - 2f(x)}{h^2} - \frac{h^2}{48}f^{(4)}(x) - \cdots \tag{3.31}$$

This leads to an approximation for the value of the second derivative of $f(x)$ at x:

$$f''(x) = 4\frac{f\left(x+\frac{h}{2}\right) + f\left(x-\frac{h}{2}\right) - 2f(x)}{h^2} + O(h^2) \tag{3.32}$$

known as the *(first) central-difference approximation* to the second derivative.

While it's not too common in physics to need derivatives beyond the second derivative, it should be straightforward to see how one would go about calculating higher-order derivatives. We've seen that sums of Taylor series give us even derivatives and differences of the Taylor series give us odd derivatives: we simply need to combine sufficiently many sums or differences, to cancel all unwanted terms. For example, $f^{(4)}(x)$ can be approximated using $f(x + h/2) + f(x - h/2)$ together with $f(x + h) + f(x - h)$.

3.3.6 Points on a Grid

So far, we've been quite cavalier in our use of different step sizes h and the placement of different points at x, $x+h/2$, $x+h$, and so on. In other words, we were taking it for granted that f was at our disposal, meaning that we could evaluate the function at any point of our choosing. This does happen sometimes in practice: for example, in section 3.5 we will evaluate a kinetic energy by numerically taking the second derivative of a wave function, with an h that is up to us. However, as we noted in the first section of the present chapter, sometimes we don't control the function and the points at which it is evaluated but, instead,

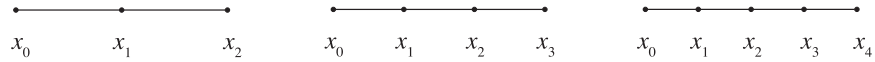


Fig. 3.3 Illustration of points and subintervals/panels

have access only to a set of n discrete data points (i.e., a table) of the form $(x_i, f(x_i))$ for $i = 0, 1, \ldots, n-1$. In this case, the function is known only at fixed points, x_i, not of our choosing.

Avoiding Error Creep

A very common use case is when the points x_i are on an equally spaced grid (also known as a *mesh*), from a to b. The n points are then given by the following relation:

$$x_i = a + ih \tag{3.33}$$

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

$$h = \frac{b-a}{n-1} \tag{3.34}$$

Thus, even if this h is small (i.e., we have many points n), it is not of our choosing.

Since we will make heavy use of the last two formulas in what follows, let us take some time to interpret them. First, note that if you use all x_i's, then you are dealing with a *closed* approach: x_i ranges from a (for $i = 0$) to b (for $i = n-1$). In other words, in that case the endpoints of our interval would be included in our set of points. Second, we are using notation that will easily translate to Python's (or any other C-based language's) 0-indexing: we start counting at 0 and stop counting at $n-1$. To be explicit: we have n points in total (two of which are the endpoints). Third, we observe that the formula for the h, Eq. (3.34), simply takes the interval from a to b and splits it up into smaller pieces. When we are dealing with n points in total, we are faced with $n-1$ subintervals from a to b. In what follows, we will be using the terms *subinterval* and *panel* interchangeably.[6] This is illustrated in Fig. 3.3; note that the overall interval (from a to b) always stays the same.

Observe that Eq. (3.33) always starts at $x = a$ and then automatically transports you to your desired x_i, e.g., if $i = 17$ it brings you up to $x_{17} = a + 17h$. You can store all of these x_i into a Python list by saying `xs = [a+i*h for i in range(n)]`. Note how `range` automatically ensures that `i` goes from `0` to `n-1`. While this may appear to be rather

[6] You might choose to introduce a new variable containing the number of panels, $N = n-1$, in which case $h = (b-a)/N$. In that case, the i in x_i would go from 0 to N, since we have $N + 1 = n$ points in total. This is perfectly legitimate, but does introduce the cognitive load of having to keep track of two different variables, n and N. We'll stick to n alone in most of what follows.

unremarkable, it's worth observing that there are those[7] who would rather use an alternative formula, namely:

$$x_{i+1} = x_i + h \tag{3.35}$$

You may now be able to see where this is going: instead of storing all the x_i's so as to use them only once, you might be tempted to simply use a "running" `x` variable, which you can keep updating inside a loop by saying `x += h`. Unfortunately, this would be a mistake: h is almost certainly bound to be evaluated with limited precision. That means that each time we are adding h to the previous result we are committing another addition error. If you are dealing with thousands of points (or more) it's easy to see how an initial error of ϵ_m in evaluating x can be exacerbated. Since we're using these points on a grid in order to evaluate a finite difference, we are setting ourselves up for failure: our `x`'s are becoming progressively worse, therefore the ordinates (i.e., the `f(x)`) will also be wrong; this is a case of "systematic creep". It's easy to see that our starting expression in Eq. (3.33) is much better: it only involves one multiplication and one addition, and is therefore always to be preferred. If you read and understood chapter 2, this point should be quite straightforward. Of course, you may choose to avoid both the systematic creep (via the use of $x_i = a + ih$) and the storing of all the x_i's, simply by reapplying $x_i = a + ih$ each time through the loop.

Finite-Difference Formulas

Now, let us assume our task is to calculate the first derivative of $f(x)$ *at the same points* x_i. To be fully explicit, let's assume we have 101 points from 0 to 5. This leads to a step size of $h = 0.05$: these points are $0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, \ldots, 4.9, 4.95, 5.0$. For example, we are interested in $f'(3.7)$ but the neighboring values at our disposal are only $f(3.65)$, $f(3.7)$, and $f(3.75)$. The forward-difference formula in Eq. (3.9):

$$f'(x) = \frac{f(x+h) - f(x)}{h} + O(h) \tag{3.36}$$

shows that if we take $x = 3.7$ we can estimate $f'(3.7)$ using $f(3.7)$ and $f(3.75)$ with $h = 0.05$. However, things are not so simple for the case of the central-difference formula in Eq. (3.24):

$$f'(x) = \frac{f\left(x + \frac{h}{2}\right) - f\left(x - \frac{h}{2}\right)}{h} + O(h^2) \tag{3.37}$$

Taking $x = 3.7$ and $h = 0.05$, this formula requires the function evaluations $f(3.675)$ and $f(3.725)$. But, as we already noted, we have access to $f(3.65)$, $f(3.7)$, and $f(3.75)$, not to $f(3.675)$ and $f(3.725)$. What we *can* do, is take the h in the central-difference formula to be twice as large, i.e., take $h = 0.1$ (this, obviously, has nothing to do with the h that was used to produce the grid of points in Eq. (3.33)). For $h = 0.1$, the central-difference formula requires $f(3.65)$ and $f(3.75)$ in order to approximate $f'(3.7)$ (and is therefore still distinct from the forward-difference formula).

Slightly generalizing our result above, the central-difference formula in Eq. (3.37) assumes that we have knowledge of $f(x + h/2)$ and $f(x - h/2)$: if all we have access to is a

[7] Especially programmers who are averse to storing quantities that they won't make much use of later on.

grid of points like that in Eq. (3.33), then we will not be able to use the central-difference formula in this form. We *will* be able to use it if we double the step size, taking $h \to 2h$:

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2) \tag{3.38}$$

In other words, by doubling the step size $h \to 2h$ used to define the equation giving us the central difference, we are able to plug in the same value of h in the formula Eq. (3.38) as that used in the forward-difference Eq. (3.36), e.g., $h = 0.05$. This happens to be a general result: all the expressions we gave above that require a function evaluation at a midpoint could be recast in a form that can use function evaluations on a grid if we simply take $h \to 2h$. For example, our equation for the central-difference approximation to the second derivative from Eq. (3.32) becomes:

$$f''(x) = \frac{f(x+h) + f(x-h) - 2f(x)}{h^2} + O(h^2) \tag{3.39}$$

which is probably easier to memorize, anyway.

Let's return to our problem of evaluating the first derivative at points on a grid x_i when all we have are the function values $f(x_i)$. We've seen that we can approximate $f'(x_i)$ using: (a) Eq. (3.36) which corresponds to the forward-difference formula with step size h, and (b) Eq. (3.38) which corresponds to the central-difference formula with step size $2h$. The more accurate method (central differrence) here uses a larger step size $2h$ and we know that a larger step size leads to less accuracy. Now, the question naturally arises: could it be that the inaccuracy stemming from the larger step size overpowers the accuracy coming from the fact that cental difference is a higher-order method?

In order to answer this question, we go back to our error analyses. For the forward difference, we already know that the total error is given by Eq. (3.15):

$$\mathcal{E} = \mathcal{E}_{app} + \mathcal{E}_{ro} = \frac{h}{2}|f''(x)| + \frac{2|f(x)|\epsilon_m}{h} \tag{3.40}$$

We will now find the corresponding expression for the new central difference. This will not be identical to Eq. (3.25), since we're now dealing with a step size of $2h$, since the formula we're analyzing is Eq. (3.38). One can analyze the new approximation from scratch, or simply replace $h \to 2h$ in Eq. (3.25) to get:

$$\mathcal{E} = \mathcal{E}_{app} + \mathcal{E}_{ro} = \frac{h^2}{6}|f'''(x)| + \frac{|f(x)|\epsilon_m}{h} \tag{3.41}$$

The trend exhibited by Eq. (3.41) appears to still be generally better than that of Eq. (3.40): the approximation term is still quadratic,[8] whereas the roundoff term is now half as big. One could always concoct artificial scenarios where $|f'''(x)|$ is dramatically larger than

[8] And with a 6 in the denominator compared to the forward difference's 2 in the denominator.

$|f''(x)|$, but for most purposes it's safe to say that the central-difference formula (even with twice the step size) is better than the forward-difference one.

As an aside, we note that if we had been less strict in our requirements for the derivative of $f(x)$, e.g., if we were OK with the possibility of getting the derivative at points other than the x_i, then we would have been able to use the unmodified central-difference formula of Eq. (3.37). To be explicit, taking $x = 3.725$ and $h = 0.05$, this formula requires the function evaluations $f(3.7)$ and $f(3.75)$ which we *do* have access to! The price to be paid, of course, is that this way we calculate $f'(3.725)$, not $f'(3.7)$.

3.3.7 Richardson Extrapolation

We now turn to a nice technique, called Richardson extrapolation, which can be used to impove the accuracy of a numerical algorithm. Here we apply it to finite-difference formulas, but its applicability is much wider, so we will return to this tool when we study integrals in chapter 7 and differential equations in chapter 8.

General Formulation

Assume your task is to evaluate the quantity G, i.e., G is the exact answer of the calculation that you are currently trying to carry out approximately. Your approximate answer $g(h)$ depends on a parameter h, which is typically an increment, or step size. We write:

$$G = g(h) + \mathcal{E}_{app}(h) \tag{3.42}$$

where we explicitly noted that the approximation error also depends on h. The procedure we are about to introduce works only on reducing the approximation error (so we didn't also include a roundoff term $\mathcal{E}_{ro}$).

We will now make the (pretty widely applicable) assumption that the error term can be written as a sum of powers of h:

$$G = g(h) + Ah^p + Bh^{p+q} + Ch^{p+2q} + \cdots \tag{3.43}$$

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 idea behind Richardson extrapolation is to apply Eq. (3.43) twice, once for a step size h and once for a step size $h/2$:

$$\begin{aligned} G &= g(h) + Ah^p + O(h^{p+q}) \\ G &= g(h/2) + A\left(\frac{h}{2}\right)^p + O(h^{p+q}) \end{aligned} \tag{3.44}$$

Equating the two right-hand-sides gives us:

$$g(h) + Ah^p = g(h/2) + A\left(\frac{h}{2}\right)^p + O(h^{p+q}) \tag{3.45}$$

This equation can now be solved for Ah^p:

$$Ah^p = \frac{2^p}{2^p - 1}\left[g(h/2) - g(h)\right] + O(h^{p+q}) \tag{3.46}$$

This result, in turn, can be plugged back into the first relation in Eq. (3.44) to give:

$$G = \frac{2^p g(h/2) - g(h)}{2^p - 1} + O(h^{p+q}) \tag{3.47}$$

This is sometimes called an *extrapolated value*. This is a good time to lay all your possible worries to rest: Eq. (3.47) contains a subtraction on the right-hand-side (so it's possible some roundoff error may creep in) but, despite the fact that $g(h)$ and $g(h/2)$ presumably have similar values, no catastrophic cancellation is present, because of the 2^p coefficient.

To summarize, we started in Eq. (3.43) from a formula that has a leading error of $O(h^p)$; by using two calculations (one for a step size h and one for a step size $h/2$), we managed to eliminate the error $O(h^p)$ and are left with a formula that has error $O(h^{p+q})$. It's easy to see how this process could be repeated: by starting with two calculations each of which has error $O(h^{p+q})$, we can arrive at an answer with error $O(h^{p+2q})$, and so on.

Finite Differences

We now apply our general result in Eq. (3.47) to finite-difference formulas. In order to bring out the connections with other approaches touched upon in this chapter, we will carry out this task twice, once for the forward difference and once for the central difference.

Forward Difference

We will use our first forward-difference formula, Eq. (3.9):

$$D_{fd}(h) = \frac{f(x+h) - f(x)}{h} \tag{3.48}$$

where we also took the opportunity to employ new notation. Clearly, the leading error term in Eq. (3.48) is $O(h)$ (as we explicitly derived in section 3.3), meaning that the exponent in h^p is $p = 1$ for this case. Richardson extrapolation will eliminate this leading error term, leaving us with the next contribution, which is $O(h^2)$, as we know from Eq. (3.7).

Applying Eq. (3.47) with the present notation for $p = 1$ gives us:

$$\begin{aligned} R_{fd} &= 2D_{fd}(h/2) - D_{fd}(h) + O(h^2) = 2\frac{f\left(x+\frac{h}{2}\right) - f(x)}{h/2} - \frac{f(x+h) - f(x)}{h} + O(h^2) \\ &= \frac{4f\left(x+\frac{h}{2}\right) - f(x+h) - 3f(x)}{h} + O(h^2) \end{aligned} \tag{3.49}$$

In the second equality we plugged in our definition from Eq. (3.48) twice. In the third equality we grouped terms together. Our result is *identical* to the second forward-difference formula from Eq. (3.27): while there we had to explicitly derive things in terms of Taylor series, here we simply carried out one step of a Richardson extrapolation process.[9] To

[9] We could now carry out two calculations with errors $O(h^2)$, set $p = 2$ in Eq. (3.47) and we would get a result with error $O(h^3)$.

summarize, we employed a two-point formula (Eq. (3.48)) twice and ended up with a three-point formula (Eq. (3.49)) which is more accurate.

Central Difference

This time we will use our first central-difference formula, Eq. (3.24):

$$D_{cd}(h) = \frac{f\left(x+\frac{h}{2}\right) - f\left(x-\frac{h}{2}\right)}{h} \tag{3.50}$$

where we used the notation $D_{cd}(h)$ notation on the left-hand side. The leading error term in Eq. (3.50) is $O(h^2)$ (as we explicitly derived in section 3.3), meaning that the exponent in h^p is $p = 2$ for this case. Richardson extrapolation will eliminate this leading error term, leaving us with the next contribution, which is $O(h^4)$, as we know from Eq. (3.28).

Applying Eq. (3.47) with the present notation for $p = 2$ gives us:

$$\begin{aligned} R_{cd} &= \frac{4}{3}D_{cd}(h/2) - \frac{1}{3}D_{cd}(h) + O(h^4) \\ &= \frac{4}{3}\frac{f\left(x+\frac{h}{4}\right) - f\left(x-\frac{h}{4}\right)}{h/2} - \frac{1}{3}\frac{f\left(x+\frac{h}{2}\right) - f\left(x-\frac{h}{2}\right)}{h} + O(h^4) \\ &= \frac{8f\left(x+\frac{h}{4}\right) + f\left(x-\frac{h}{2}\right) - f\left(x+\frac{h}{2}\right) - 8f\left(x-\frac{h}{4}\right)}{3h} + O(h^4) \end{aligned} \tag{3.51}$$

In the second line we plugged in our definition from Eq. (3.50) twice. In the third line we grouped terms together. Our result is *identical* to the second central-difference formula that you were asked to derive after Eq. (3.28): while there we had to explicitly derive things in terms of Taylor series, here we simply carried out one step of a Richardson extrapolation process. If you're not seeing that the two results are identical, take $h \to 2h$ in the present result. To summarize, we employed a two-point formula (Eq. (3.50)) twice and ended up with a four-point formula (Eq. (3.51)) which is more accurate.[10]

Implementation

Code 3.2 is an implementation of Richardson extrapolation: crucially, this works by combining the outputs of our earlier forward- and central-difference functions. The main new feature in this code is that we have opted against copying and pasting functions from an earlier file: that would be needlessly error-prone. Instead, what we've done here is to import specific functions from `finitediff.py` which take care of the function itself, its analytical derivative, the forward difference, and the central difference. In addition to making the present code shorter, this has the added advantage that if we ever update those functions, we only need to change them in one location, namely in the file where they are defined.[11] The output of running this code is:

[10] Again, we could now carry out two calculations with errors $O(h^4)$, set $p = 4$ in Eq. (3.47) and we would get a result with error $O(h^6)$.

[11] As long as we don't break the interface, i.e., if we keep the same input and output conventions.

Code 3.2 richardsondiff.py

```
from finitediff import f, fprime, calc_fd, calc_cd

x = 0.5
an = fprime(x)

hs = [10**(-i) for i in range(1,7)]

rowf = "{0:1.0e} {1:1.16f} {2:1.16f}"
print("h    abs. err. rich fd     abs. err. rich cd")
for h in hs:
     fdrich = 2*calc_fd(f,x,h/2) - calc_fd(f,x,h)
     fd = abs(fdrich-an)
     cdrich = (4*calc_cd(f,x,h/2) - calc_cd(f,x,h))/3
     cd = abs(cdrich-an)
     print(rowf.format(h,fd,cd))
```

```
h     abs. err. rich fd  abs. err. rich cd
1e-01 0.0259686059827384 0.0000098728371007
1e-02 0.0002695720500450 0.0000000009897567
1e-03 0.0000027005434182 0.0000000000009619
1e-04 0.0000000270109117 0.0000000000043667
1e-05 0.0000000003389138 0.0000000000132485
1e-06 0.0000000006941852 0.0000000002500959
```

Starting from the forward-difference Richardson-extrapolated results: every time we reduce the h by an order of magnitude, the absolute error is reduced by two orders of magnitude, consistent with a method that has an approximation error of $O(h^2)$, see Eq. (3.49). This process reaches a minimum at $h = 10^{-5}$: the absolute error there is $\approx 3 \times 10^{-10}$. Note that this behaviour is very similar to the output of `finitediff.py` *for the central-difference column*, which also resulted from a method with an error of $O(h^2)$. The last line in the present output merely serves to show that roundoff has started dominating the forward-difference case, so we can no longer rely on the Richardson extrapolation process.

The central-difference Richardson-extrapolated results are analogous. The first time we reduce h by an order of magnitude we reduce the absolute error by a whopping *four* orders of magnitude, consistent with a method that has an approximation error of $O(h^4)$, see Eq. (3.51). This cannot continue indefinitely: in the next step we improve by three orders of magnitude, reaching a minimum at $h = 10^{-3}$: the absolute error there is $\approx 10^{-12}$, the best we've seen so far using a finite-difference(-related) scheme. As the h is further reduced roundoff error starts dominating, so the extrapolation process is no longer reliable.

3.4 Automatic Differentiation

Up to this point in this chapter, we have seen that there are two ways of taking derivatives on a computer: first, *analytically* using a symbolic algebra package or library and, second, using *finite-difference* formulas of varying sophistication. While this is OK for pedagogical clarity, it's not quite true. There exists a third option, known as *automatic differentiation* or, sometimes, *algorithmic differentiation*.

Qualitatively, automatic differentiation is equivalent to analytical differentiation of elementary functions along with propagation using the chain rule. Crucially, this is *not* accomplished via the manipulation of expressions (as is done in symbolic differentiation, which is quite inefficient in practice) but using specific numbers. Thus, we never have access to an analytical expression providing the derivative of a function, but do get a result with nearly machine precision for the derivative at any one point. Thus, automatic differentiation is considerably more accurate than finite differences: it's as accurate as symbolic differentiation, but it doesn't need to produce a general expression (which is, anyway, only going to be used at specific points).

Using Python, one has access to several implementations of automatic differentiation, e.g., JAX: the details are strongly time dependent so we will, instead, focus on the main idea. (One problem asks you to experiment with `jax`, whereas another problem guides you toward building a bare-bones automatic differentiator yourself.) In the past, computational physics books did not mention automatic differentiation but, given its benefits and conceptual simplicity, it deserves a wider audience.

3.4.1 Dual Numbers

Extend any number a by also adding a second component:

$$\mathbf{a} = a + a'd \tag{3.52}$$

These are called *dual numbers*. This d is simply a placeholder telling us what the second component, a', is. This is analogous to complex numbers, where we go from x to $x + yi$ via the use of the imaginary unit, $i = \sqrt{-1}$. Here, we take $d^2 = 0$ (compare with $i^2 = -1$). If you're uncomfortable with the possibility of a number being non-zero but giving zero when squared, look up the term *Grassmann variable*.

It is easy to see how arithmetic works for dual numbers. For example, for addition:

$$\mathbf{a} + \mathbf{b} = (a + a'd) + (b + b'd) = a + b + (a' + b')d \tag{3.53}$$

Similarly, for multiplication we have:

$$\mathbf{a} \times \mathbf{b} = (a + a'd) \times (b + b'd) = ab + ab'd + a'bd + a'b'd^2 = ab + (ab' + a'b)d \tag{3.54}$$

To go to the third equality we made use of the fact that $d^2 = 0$. You can see how subtraction and division will turn out. What is emerging here is an arithmetic where the first component behaves as real numbers do, whereas the second component follows well-known rules of

differentiation (for the sum, the product, etc.). At this point, we can drop the use of d entirely, by rewriting dual numbers using ordered pairs of real numbers:

$$\mathbf{a} = (a, a') \tag{3.55}$$

As was implicit above, a gives us the value of a function at a specific point and a' the value of the derivative of that function at the same point. This definition using an ordered pair is completely analogous to complex numbers, which are also given as $z = (x, y)$. We can now re-express (and augment) the basic arithmetic rules as follows:

$$\begin{aligned}
\mathbf{a} + \mathbf{b} &= (a + b, a' + b') \\
\mathbf{a} - \mathbf{b} &= (a - b, a' - b') \\
\mathbf{a} \times \mathbf{b} &= (ab, ab' + a'b) \\
\mathbf{a} \div \mathbf{b} &= \left(\frac{a}{b}, \frac{a'b - ab'}{b^2}\right)
\end{aligned} \tag{3.56}$$

Thus, dual numbers give us a way to differentiate elementary mathematical operations. In actual calculations, we'll need to also know how to treat constants, A, and independent variables, x. Simply define:

$$\begin{aligned}
\mathbf{A} &= (A, 0) \\
\mathbf{x} &= (x, 1)
\end{aligned} \tag{3.57}$$

Both of these are plausible, since the derivative of a constant is 0 and the derivative of x with respect to x is 1. In practice, we'll be faced with a function $f(x)$ for which we would like to produce the derivative. To accomplish this, we replace all occurences of x with $\mathbf{x}$, all occurences of constants A with $\mathbf{A}$, and also employ the basic arithmetic rules in Eq. (3.56). Thus, we arrive at a new function $\mathbf{f}(\mathbf{x})$: evaluating this at the specific point $(x_0, 1)$ we get the ordered pair $(f(x_0), f'(x_0))$, thereby achieving what we set out to do.

3.4.2 An Example

Let's apply all of the above rules to a specific example. Take:

$$f(x) = \frac{(x - 2)(x - 3)}{x - 4} \tag{3.58}$$

We are interested in computing $f(6)$ and $f'(6)$. We can immediately see (by plugging in) that $f(6) = 6$. However, the value of $f'(6)$ is not as obvious. In order to find it, we employ the above prescription, promoting all numbers to dual numbers and interpreting the arithmetic operations appropriately. We have:

$$\begin{aligned}
\mathbf{f}(\mathbf{x}) &= (\mathbf{x} - \mathbf{2}) \times (\mathbf{x} - \mathbf{3}) \div (\mathbf{x} - \mathbf{4}) \\
&= [(x, 1) - (2, 0)] \times [(x, 1) - (3, 0)] \div [(x, 1) - (4, 0)]
\end{aligned} \tag{3.59}$$

Now, here's the beauty of it all. Without needing to find an expression for f', we will calculate the derivative at the specific point $\mathbf{x} = (6, 1)$ by plugging in and using the rules:

$$\begin{aligned}\mathbf{f}((6,1)) &= [(6,1) - (2,0)] \times [(6,1) - (3,0)] \div [(6,1) - (4,0)] \\ &= (4,1) \times (3,1) \div (2,1) = (12,7) \div (2,1) = \left(6, \frac{1}{2}\right)\end{aligned} \tag{3.60}$$

In the second equality we simply did all the subtractions. In the third equality we did the multiplication. In the fourth equality we carried out the division. This gives us $f(6) = 6$ (which we had already figured out) and $f'(6) = 1/2$ (which we hadn't). As advertised, we have arrived at the derivative of our function at a specific point simply by plugging in values and following the rules above, without having to produce an expression for f'.

3.4.3 Special Functions

With a view to extending this formalism, let's start with a polynomial:

$$p(a) = c_0 + c_1 a + c_2 a^2 + c_3 a^3 + \cdots + c_{n-1} a^{n-1} \tag{3.61}$$

We now wish to promote a to $\mathbf{a}$: we'll need to know how to handle powers of $\mathbf{a}$. These are easy to calculate via repeated use of the multiplication rule from Eq. (3.56):

$$\mathbf{a}^2 = \mathbf{a} \times \mathbf{a} = (a^2, 2aa') \tag{3.62}$$

Similarly:

$$\mathbf{a}^3 = \mathbf{a}^2 \times \mathbf{a} = (a^3, 3a^2 a') \tag{3.63}$$

and so on. Thus, promoting the a to (a, a') and the constants c_i to $(c_i, 0)$ gives us:

$$\begin{aligned}\mathbf{p}(\mathbf{a}) &= (c_0,0) + (c_1,0) \times \mathbf{a} + (c_2,0) \times \mathbf{a}^2 + (c_3,0) \times \mathbf{a}^3 + \cdots + (c_{n-1},0) \times \mathbf{a}^{n-1} \\ &= (c_0,0) + (c_1 a, c_1 a') + (c_2 a^2, c_2 2aa') + \cdots + (c_{n-1} a^{n-1}, c_{n-1}(n-1)a^{n-2}a') \\ &= (c_0 + c_1 a + c_2 a^2 + \cdots + c_{n-1} a^{n-1}, c_1 a' + c_2 2aa' + \cdots + c_{n-1}(n-1)a^{n-2}a') \\ &= (p(a), a' p'(a))\end{aligned} \tag{3.64}$$

In the second line we carried out all the multiplications. In the third line we did the summations. In the fourth line we identified in the second component the presence of p', namely the derivative of the polynomial. This result, which we explicitly proved for polynomials, leads us to think about other functions, since we will also need to know how to take, say, the cosine of a dual number. We define the relevant (chain) rule as:

$$\mathbf{g}(\mathbf{a}) = \mathbf{g}((a, a')) = (g(a), a' g'(a)) \tag{3.65}$$

which looks completely analogous to our result for polynomials in Eq. (3.64) above. Note that the $'$ on the right-hand side of this equation has two distinct (though related) meanings:

a' is the second component of $\mathbf{a}$, whereas g' refers to the derivative of the function g , which is arrived at through some other means. A few examples:

$$\begin{aligned}
\mathbf{sin}(\mathbf{a}) &= \mathbf{sin}\left((a, a')\right) = (\sin a, a' \cos a) \\
\mathbf{cos}(\mathbf{a}) &= \mathbf{cos}\left((a, a')\right) = (\cos a, -a' \sin a) \\
\mathbf{e^a} &= \mathbf{e}^{(a,a')} = (e^a, a' e^a) \\
\mathbf{ln}(\mathbf{a}) &= \mathbf{ln}\left((a, a')\right) = \left(\ln a, \frac{a'}{a}\right) \\
\mathbf{sqrt}(\mathbf{a}) &= \mathbf{sqrt}\left((a, a')\right) = \left(\sqrt{a}, \frac{a'}{2\sqrt{a}}\right)
\end{aligned} \tag{3.66}$$

We are using new symbols on the left-hand sides, in the spirit in which above we promoted the function f to $\mathbf{f}$ (or the function g to $\mathbf{g}$). These can either be seen as results of Eq. (3.65) or could be explicitly derived via the relevant Taylor expansion each time.

A software system that implements automatic differentiation knows how to apply all these formulas, even if it needs to apply more than one to the same expression. To make this crystal clear, we now turn to our usual example, $f(x) = e^{\sin(2x)}$ at $x = 0.5$. Following the rules of promotion, as above, we have:

$$\mathbf{f}(\mathbf{x}) = \mathbf{e}^{\mathbf{sin}((2,0)\times(x,1))} \tag{3.67}$$

We now plug in $x = 0.5$ to find:

$$\mathbf{f}\left((0.5, 1)\right) = \mathbf{e}^{\mathbf{sin}((2,0)\times(0.5,1))} = \mathbf{e}^{\mathbf{sin}((1,2))} = \mathbf{e}^{(\sin(1), 2\cos(1))} = \left(e^{\sin(1)}, 2\cos(1)e^{\sin(1)}\right) \tag{3.68}$$

In the second equality we carried out the multiplication as per Eq. (3.56). In the third equality we used the chain rule in Eq. (3.66) for the sine. In the fourth equality we used the chain rule in Eq. (3.66) for the exponential. Thus, we have arrived at the output $f(0.5) = e^{\sin(1)} \approx 2.319\,776\,824\,715\,853$ and $f'(0.5) = 2\cos(1)e^{\sin(1)} \approx 2.506\,761\,534\,986\,894$. These results agree (to within machine precision) with the answer we get by analytically evaluating the derivative and then plugging in $x = 0.5$.

It's important to understand that the procedure we followed was not actually that of taking a derivative. We were merely dealing with numbers, to which we applied the promotions to ordered pairs as per Eq. (3.57), the basic arithmetic rules in Eq. (3.56), and the chain rule in Eq. (3.65): the result turns out to be a derivative, even though the elementary operations involved simply the additions, divisions, etc. of real numbers.

Before concluding, it's worth underlining that the procedure we followed for automatic differentiation doesn't exactly address one of our main tasks, namely the case where we have access only to a table of points $(x_i, f(x_i))$ for $i = 0, 1, \ldots, n-1$.[12] Instead, automatic differentiation can be applied to the piece of code producing a function evaluation at one point: it then promotes all numbers to ordered pairs and follows simple rules to evaluate the function's derivative at the same point, without any major overhead.

[12] For that matter, symbolic differentiation, as discussed in section 3.2 doesn't either, except if one has first interpolated and then analytically differentiates the interpolating function.

3.5 Project: Local Kinetic Energy in Quantum Mechanics

We will now see how taking derivatives is an essential part of evaluating the kinetic energy in quantum mechanics (in the position basis). If you haven't studied quantum mechanics, you can skim through most of section 3.5.1. Even if you don't know what a wave function is, there are some lessons to be learned implementation-wise: we will show how one Python function can handle different physical scenarios, as long as we are careful to respect some general conventions about function-interface design.

Physics-wise, our coverage will revolve around one non-relativistic particle; we will study two different interaction terms (harmonic oscillator and free particle). To refresh your memory, the *time-independent Schrödinger equation* involves the Hamiltonian $\hat{H}$, the wave function ψ, and the energy E:

$$\hat{H}\psi = E\psi \tag{3.69}$$

This is an *eigenvalue problem*, about which we'll have a lot more to say in the rest of this volume, especially in chapter 4 (where we discuss linear algebra) and chapter 8 (where we study differential equations). In general, the Hamiltonian operator is made up of the kinetic energy operator and the potential energy operator: $\hat{H} = \hat{T} + \hat{V}$.

3.5.1 Single-Particle Wave Functions in One Dimension

We now go over some standard results, assuming that you've encountered this material before, so we will not repeat the explicit derivations.

Quantum Harmonic Oscillator

An important problem in one-dimensional single-particle quantum mechanics is the harmonic oscillator; for this case, the time-independent Schrödinger equation takes the form:

$$-\frac{\hbar^2}{2m}\frac{d^2\psi(x)}{dx^2} + \frac{1}{2}m\omega^2 x^2\psi(x) = E\psi(x) \tag{3.70}$$

where $\hbar$ is Planck's constant (divided by 2π) and the particle's mass is m. The left-hand side is made up of the kinetic energy (involving a second derivative) and the potential energy (involving a term quadratic in x). The ω is the (classical) angular frequency.

Every quantum mechanics textbook discusses how to solve this problem. This means how to determine the energy eigenvalues:

$$E_n = \left(n + \frac{1}{2}\right)\hbar\omega \tag{3.71}$$

and the (normalized) eigenfunctions:

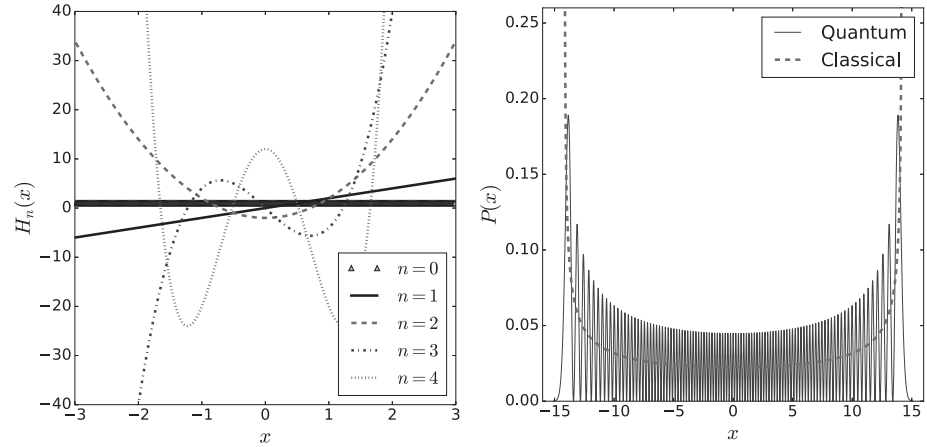


Fig. 3.4 First few Hermite polynomials (left) and harmonic oscillator probability density (right)

$$\psi_n(x) = \frac{1}{\sqrt{2^n n!}} \left(\frac{m\omega}{\pi\hbar}\right)^{1/4} H_n\left(\sqrt{\frac{m\omega}{\hbar}}x\right) e^{-m\omega x^2/(2\hbar)} \tag{3.72}$$

Note that both the eigenvalues and the eigenfunctions are labelled by a discrete index n (a non-negative integer): this goes to the heart of *quantization*. Notoriously, the ground state of the harmonic oscillator is $n = 0$, which is characterized by a finite energy as per Eq. (3.71); this is known as *zero-point motion*.

For a given n, the harmonic-oscillator eigenfunction is equal to (some prefactors times) a Gaussian term and another term involving an *Hermite polynomial*, $H_n(x)$. Hermite polynomials can be evaluated via a process similar to that introduced in section 2.5 for Legendre polynomials. They obey the *recurrence relation*:

$$H_{j+1}(x) = 2xH_j(x) - 2jH_{j-1}(x) \tag{3.73}$$

To step through this process, one starts with the known first two polynomials, $H_0(x) = 1$ and $H_1(x) = 2x$, and calculates $H_n(x)$ by taking:

$$j = 1, 2, \ldots, n-1 \tag{3.74}$$

The first few Hermite polynomials are shown in the left panel of Fig. 3.4, which is analogous to the left panel of Fig. 2.6. Similarly, for the derivative one can use the relation:

$$H_n'(x) = 2nH_{n-1}(x) \tag{3.75}$$

though we don't need to evaluate these derivatives here (we will later on). To summarize, for a specified n the wave function is given by Eq. (3.72), which you can immediately compute if you know how to evaluate Hermite polynomials numerically.

Correspondence Principle

Let us recall the harmonic oscillator in *classical* mechanics. If the amplitude of the oscillation is x_0, then the total energy E, which is a constant of the motion, is:

$$E = \frac{1}{2} m\omega^2 x_0^2 \tag{3.76}$$

This describes the extreme case where the kinetic energy is zero; there are two such *turning points*, $\pm x_0$. As you may recall, the particle oscillates between them, $-x_0 \leq x \leq x_0$. The regions $x^2 > x_0^2$ are *classically forbidden*, since they correspond to negative kinetic energy. (Things are different in quantum mechanics, since there the wave function may "leak" into the classically forbidden region.) A problem guides you toward deriving the following equation for the *classical probability density* for the oscillator problem:

$$P_c(x) = \frac{1}{\pi \sqrt{x_0^2 - x^2}} \tag{3.77}$$

This is the probability that we will find the particle in an interval dx around the position x; it is larger near the turning points: that's where the speed of the particle is smallest.

A fascinating question arises regarding how we go from quantum to classical mechanics; this is described by what is known as the *correspondence principle*: as the quantum number n goes to infinity, we should recover classical behavior. In order to see this, the right panel of Fig. 3.4 plots results for the quantum harmonic oscillator for the case of $n = 100$, which is reasonably large. The solid curve, exhibiting many wiggles, is the probability density for the quantum case; as per the *Born rule*, this is the square of the modulus of the wave function, i.e., $P(x) = |\psi(x)|^2$. To compare apples to apples, we also study a classical problem (dashed curve) with an amplitude of:

$$x_0 = \sqrt{\frac{\hbar}{m\omega}(2n+1)} \tag{3.78}$$

which we found by equating Eq. (3.76) with Eq. (3.71). As the figure clearly shows, if we take a local average of the quantum probability density in a small interval around x (essentially smearing out the wiggles), we get something that is very similar to the classical probability density. This is the correspondence principle in action.

Particle in a Periodic Box

Another standard example covered in introductory quantum-mechanics courses is that of a "particle in a box"; what is usually meant by this is a particle that is confined inside an infinite potential well, i.e., one for which the wave function must go to zero at the edges. Here, we study a different scenario, i.e., a *periodic* box, which is usually important in the study of extended (i.e., non-confined) matter. Our box goes from $-L/2$ to $L/2$ (i.e., has a length of L); inside it, the time-independent Schrödinger equation is simply:

$$-\frac{\hbar^2}{2m}\frac{d^2\psi(x)}{dx^2} = E\psi(x) \tag{3.79}$$

since we're assuming there is no external potential inside the box. The eigenfunctions are simply plane waves, i.e., complex exponentials:

$$\psi_k(x) = \frac{1}{\sqrt{L}} e^{ikx} \tag{3.80}$$

The prefactor is such that our eigenfunction is normalized. We have labelled our eigenfunctions with the wave number k; it is related to the momentum via the relation $p = \hbar k$.

In order to find out more about this wave number/momentum, let us investigate the boundary conditions. We mentioned above that we are dealing with a *periodic* box; mathematically, this is expressed by saying $\psi_k(x) = \psi_k(x + L)$, which is precisely the relation that allowed us to stay inside a single box in the first place ($-L/2 \leq x \leq L/2$): the wave function simply repeats itself beyond that point. If we now combine this periodicity condition with our plane waves of Eq. (3.80), we find:

$$k = \frac{2\pi}{L} n \tag{3.81}$$

where n is an integer which could be positive, negative, or zero. This is already different from the case of the harmonic oscillator above. If we know the value of this quantum number n, i.e., we know the value of the wave number for a given state, then we are also able to calculate the energy corresponding to that state; to see this, plug the eigenfunction Eq. (3.80) into the eigenvalue equation Eq. (3.79) to find:

$$E = \frac{\hbar^2 k^2}{2m} = \frac{\hbar^2}{2m} \left(\frac{2\pi}{L} \right)^2 n^2 \tag{3.82}$$

In the second equality we used our result in Eq. (3.81). As expected for this problem (for which there is no attraction and therefore no bound state), the energy cannot be negative.

Implementation

Let us see how to implement our two possibilities for the wave function. The task at hand is pretty mundane; in order to make it a bit interesting, we have decided to write two functions *with the same interface*; that means that the external world will treat wave function code the same way, without caring about details of the implementation.

Both of our functions will take in two parameters: first, the position of the particle and, second, a dictionary bundling together any wave-function-specific parameters. This is not idle programming ("code golf"): as you will discover in the following section, a common interface makes a world of difference when you need to pass these two functions into a *third* function. We implement a general solution, similar to Python's `**kwargs`, to help students who are still becoming comfortable with dictionaries in Python.

Code 3.3 starts with import statements, as usual. We will need an exponential that handles real numbers (for the harmonic oscillator) and another one that can handle complex numbers (for the plane wave of the box problem). To avoid a name clash, we import `exp` explicitly and then import the `cmath` module, from which we intend to use the complex

psis.py Code 3.3

```
from math import sqrt, pi, factorial, exp
import cmath

def hermite(n,x):
     val0 = 1.; val1 = 2*x
     for j in range(1,n):
          val2 = 2*x*val1 - 2*j*val0
          val0, val1 = val1, val2
     dval2 = 2*n*val0
     return val2, dval2

def psiqho(x,nametoval):
     n = nametoval["n"]
     momohbar = nametoval["momohbar"]
     al = nametoval["al"]
     psival = momohbar**0.25*exp(-0.5*al*momohbar * x**2)
     psival *= hermite(n,sqrt(momohbar)*x)[0]
     psival /= sqrt(2**n * factorial(n) * sqrt(pi))
     return psival

def psibox(x,nametoval):
     n = nametoval["n"]
     boxl = nametoval["boxl"]
     return cmath.exp(2*pi*n*x*1j/boxl)

if __name__ == '__main__':
     x = 1.
     nametoval = {"n": 100, "momohbar": 1., "al": 1.}
     psiA = psiqho(x, nametoval)
     nametoval = {"n": -2, "boxl": 2*pi}
     psiB = psibox(x, nametoval)
     print(psiA, psiB)
```

exponential `cmath.exp()`. Our code then defines a function that evaluates Hermite polynomial values and derivatives. This is very similar to our earlier `legendre()`, only this time we don't cater to the possibiities $n = 0$ or $n = 1$, in order to shorten the code.

Our first wave function, `psiqho()`, takes in a float and a dictionary, as advertised. This implements Eq. (3.72): in addition to the position of the particle, we also need to pass in

the quantum number n, as well as the value of $m\omega/\hbar$. To spice things up, we also allow for the possibility of including an extra variational parameter, α, in the exponent of the Gaussian. That brings the total of extra parameters needed up to three. The dictionary `nametoval` is aptly named: it maps from a string (the name) to a value; thus, by passing in the appropriately initialized (in the external world) dictionary, we manage to bundle together all the necessary extra parameters, while still keeping our interface clean. The rest of the function is pretty straightforward.

Our second wave function, `psibox()`, also takes in a float and a dictionary. As before, the float is for the position of the particle and the dictionary for any other parameters needed to evaluate the wave function for this case. To implement Eq. (3.80), we need to pass in, first, the quantum number and, second, the length of the box side, L. Note how different the requirements of our two wave functions are: `psiqho()` takes in a position and then three more numbers, whereas `psibox()` accepts a position and then two more numbers; despite having different parameter needs, the two functions have the same interface, due to our decision to bundle the "extra" parameters into a dictionary. Of course, inside `psibox()` we immediately "unpack" this dictionary to get the values we need. We then use `cmath.exp()`.

The main program explicitly defines a `nametoval` dictionary that bundles the extra parameters for the oscillator problem, before passing it to `psiqho()`. The test code calling `psibox()` is analogous: a `nametoval` dictionary is defined; note that, while, the two dictionaries look quite different, the parts of the code *calling* `psiqho()` and `psibox()` are identical. The output of this code is not very important here; the main point is the common interface, which will help us below.

3.5.2 Second Derivative

You may have realized that the code we just discussed evaluates the wave functions *only*, i.e., it has nothing to do with energies. We will now see how to use that code to evaluate the kinetic energy, by implementing the second derivative. This will emphasize the aforementioned point on interface design, but will also be of wider applicability.

Before we discuss how to code things up, we first see *why* someone might want to evaluate the kinetic energy. Didn't we already solve this problem by finding the eigenenergies in Eq. (3.71) and Eq. (3.82)? Notice that here we're not trying to find the *total energy*, only the *kinetic energy*; note, further, that we're not trying to evaluate the *expectation value* of the kinetic energy but, instead, only the value of the kinetic energy *for a given position*. Motivated by Eq. (3.69), $\hat{H}\psi = E\psi$, we define the following quantity, known as the *local kinetic energy*:

$$T_L = \frac{\hat{T}\psi}{\psi} \tag{3.83}$$

In general, the value of T_L will depend on x. The normalization of the wave function doesn't matter: it appears in both numerator and denominator, and cancels out.

Of course, for the simple cases studied here we've actually solved the problem of evaluating T_L, too: first, for the periodic box, there is no potential energy term, so the kinetic energy *is* the total energy; also, as per Eq. (3.82), for this specific case the energy does not depend on the position of the particle. Second, for the one-dimensional harmonic oscillator problem, given that $\hat{T}\psi + \hat{V}\psi = E\psi$, and the eigenenergy is given by Eq. (3.71), we find:

$$T_L = \frac{\hat{T}\psi}{\psi} = \left(n + \frac{1}{2}\right)\hbar\omega - \frac{1}{2}m\omega^2 x^2 \tag{3.84}$$

In other words, you can analytically determine T_L once you know all the quantities on the right-hand side. Obviously, we can find a local kinetic energy that is *negative*: just pick x to be large enough in magnitude; this is a situation that was not allowed in classical mechanics.

Even though the problem here is already solved ahead of time, the tools we will develop will be much more general, and therefore applicable also to situations where the wave function ψ is, say, not analytically known. Similarly, in the examples above we were studying *eigenfunctions*, for which the corresponding eigenvalues were also known; the approach that implements the local kinetic energy (i.e., basically, the second derivative) is general enough not to care whether or not your ψ is an eigenfunction. Finally, as we'll see in the Project of chapter 7, the local kinetic energy also arises in quantum Monte Carlo calculations for *many*-particle systems, so what we discuss here is part of the scaffolding for that, more involved, problem.

Looking at the Schrödinger equations for our problems, Eq. (3.70) and Eq. (3.79), we see that they involve the second derivative (times $-\hbar^2/(2m)$). We now turn to the main theme of this chapter, namely the evaluation of derivatives via finite differences:

$$\frac{d^2\psi(x)}{dx^2} \approx \frac{\psi(x+h) + \psi(x-h) - 2f(x)}{h^2} \tag{3.85}$$

as per Eq. (3.39).

Implementation

Code 3.4 is an implementation of the local kinetic energy. We start by importing the two wave functions from Code 3.3. The core of the new program is the function `kinetic()`. Its parameters are a wave function, a position, the infamous dictionary for extra parameters, and then an optional step size h for the finite differencing. This kinetic function interoperates with any wave function that respects our common interface (described above). We start by setting $\hbar^2/m = 1$ for simplicity. We then proceed to carry out the three function evaluations shown in Eq. (3.85). Note that we are not changing the value of x, but are employing "shifted" positions directly, wherever we need them. If you, instead, say something like `x += h`, you may be in for a surprise: as we learned in chapter 2, steps like $x + h$ followed by $x - 2h$ followed by $x + h$ don't always give back x, as there may be roundoff error accumulation. The rest of the function is quite straightforward.

Next, we define a *test function*: this contains the type of code you may have been putting in the main program so far (as did we, say, in Code 3.3). This is an example of standard

Code 3.4 kinetic.py

```
from psis import psiqho, psibox
from math import pi

def kinetic(psi,x,nametoval,h=0.005):
     hom = 1.
     psiold = psi(x,nametoval)
     psip = psi(x+h,nametoval)
     psim = psi(x-h,nametoval)

     lapl = (psip + psim - 2.*psiold)/h**2
     kin = -0.5*hom*lapl/psiold
     return kin

def test_kinetic():
     x = 1.
     hs = [10**(-i) for i in range(1,6)]
     nametoval = {"n": 100, "momohbar": 1., "al": 1.}
     qhos = [kinetic(psiqho,x,nametoval,h) for h in hs]
     nametoval = {"n": -2, "boxl": 2*pi}
     boxs = [kinetic(psibox,x,nametoval,h) for h in hs]

     rowf = "{0:1.0e} {1:1.16f} {2:1.16f}"
     print("h     qho         box")
     for h,qho,box in zip(hs,qhos,boxs):
          print(rowf.format(h,qho,box))

if __name__ == '__main__':
     test_kinetic()
```

programming practice: by encapsulating your test code here, you are free to later define *other* test functions that probe different features. The body of the function itself is not too exciting: we define a list of h's for which we wish to evaluate the (finite-difference approximation to the) second derivative, and then we call `kinetic()`. Each time, we define the appropriate extra-parameter dictionary, and then pass in the appropriate wave function to `kinetic()`. This is followed by a nicely formatted print-out of the values we computed. The main program then does nothing other than call our one test function (which in its turn calls `kinetic()`, which goes on to call the wave functions repeatedly).

The output of running this code is:

```
h      qho                  box
1e-01 83.7347487381334759 1.9933422158758425+0.0000000000000063j
1e-02 99.8252144561287480 1.9999333342231158+0.0000000000004808j
1e-03 99.9982508976935520 1.9999993333621850-0.0000000001525828j
1e-04 99.9999829515395646 1.9999999988482300-0.0000000002804216j
1e-05 99.9997983578354876 1.9999999227612262+0.0000011797609070j
```

In order to interpret this output, we first note that, for our chosen input parameters, we analytically expect the oscillator answer (see Eq. (3.84)) to be 100, while the box answer should be 2 (see Eq. (3.82)). As the step size gets smaller, we get an increasingly good approximation to the second derivative of the wave function; of course, as we keep reducing h at some point roundoff error catches up to us, so further decreases are no longer helpful. For the oscillator, from the right panel of Fig. 3.4 we immediately see that if we wish to have any hope of capturing the behavior of the wave function for $n = 100$, then we had better ensure that the step size is much smaller than the wiggles. Next, we notice that the box answer comes with an imaginary part, which Eq. (3.82) tells us should be zero: as the real part becomes better, the imaginary part becomes larger, so one would have to balance these two properties (and drop the imaginary part).

3.6 Problems

1. We will study a simple quadratic polynomial:

$$f(x) = ax^2 + bx + c \tag{3.86}$$

   Apply the forward-difference formula, Eq. (3.9), and the central-difference approximation, Eq. (3.24), and compare with the analytically known derivative:

$$f'(x) = 2ax + b \tag{3.87}$$

   You should do this by hand, i.e., without any programming. Explain your findings using the Taylor-series arguments that we encountered in the main text.
2. For this problem you will need to make sure `SymPy` is installed on your system. Use `SymPy` to analytically differentiate the function $f(x) = e^{\sin(2x)}$. Then, evaluate the derivative at a few points from $x = 0$ to $x = 0.5$. Compare the latter values to the `fprime()` we encountered in the main text.
3. Justify Eq. (3.25) and then use it to show Eq. (3.26).
4. Create a function that takes in three lists and produces the output in Fig. 3.2. Feel free to use the `xscale()` and `yscale()` functions.
5. Modify `finitediff.py` to also plot the second forward-difference and the second-central difference results. Comment on whether the plot agrees with the analytical expectation coming from the approximation-error dependence on h.

6. Show Eq. (3.27) by combining the Taylor series for $f(x+h)$ and $f(x+h/2)$. Then, show Eq. (3.28) by subtracting two pairs of Taylor series (pairwise): $f(x+h/2)$ and $f(x-h/2)$, on the one hand, $f(x+3h/2)$ and $f(x-3h/2)$, on the other.
7. This problem deals with the error behaviour of the first central-difference approximation to the second derivative.

   (a) Start with the error analysis, including both approximation and roundoff error. Derive expressions for the h_{opt} and the $\mathcal{E}_{opt}$. Then, produce numerical estimates for h_{opt} and the $\mathcal{E}_{opt}$. Compare these results to those for the first derivative.
   (b) Now code this problem up in Python (for the function $f(x) = e^{\sin(2x)}$ at $x = 0.5$) to produce both a table of numbers and a plot for the absolute error, with h taking on the values 10^{-1}, 10^{-2}, 10^{-3}, …, 10^{-10}.

8. This problem studies the second derivative of the following function:

$$f(x) = \frac{1-\cos x}{x^2} \tag{3.88}$$

   which you encountered in one of the problems in chapter 2. We take $x = 0.004$.

   (a) Start by analytically evaluating the second derivative, and plugging in $x = 0.004$.
   (b) Let h take on the values $10^{-1}, 10^{-2}, \ldots, 10^{-6}$ and produce a log-log plot of the absolute error in the first central-difference approximation to the second derivative.
   (c) Introduce a new set of points to the previous plot, this time evaluating the first central-difference approximation to the second derivative not of $f(x)$, but of the analytically rewritten version you arrived at in the previous chapter (recall: you had used a trigonometric identity which enabled you to avoid the cancellation).

9. Similarly to what we did in the main text for the second derivative, derive the first central-difference formula for the third derivative.
10. In section 2.5 we introduced Legendre polynomials via Eq. (2.81), the generating function which led to a recurrence relation, Eq. (2.86). Here we discuss another representation of Legendre polynomials, that produced by *Rodrigues' formula*:

$$P_n(x) = \frac{1}{2^n n!}\frac{d^n}{dx^n}\left[(x^2-1)^n\right] \tag{3.89}$$

    (a) Analytically derive the leading coefficient of the Legendre polynomial:

$$a_n = \frac{(2n)!}{2^n (n!)^2} \tag{3.90}$$

    This a_n is the coefficient multiplying x^n when you expand $P_n(x)$ out.
    (b) For $h = 0.01$ employ Eq. (3.24) and Rodrigues' formula to compute $P_1(x)$. Then, compute the central-difference approximation to the second derivative (as the central difference of the central difference) to compute $P_2(x)$. Keep calling the earlier central-difference function(s) and go up to $P_8(x)$. Each time, you should plot your function against the output of `legendre.py`.
    (c) What happens if you change h to 10^{-3}?

11. This problem deals with our analytical expectations for the total error, $\mathcal{E} = \mathcal{E}_{app} + \mathcal{E}_{ro}$, for Eq. (3.40) and Eq. (3.41). Let h take on the values $10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}$ and plot the total error for the two cases. Assume $f(x) = 1$, $f''(x) = 0.1$, and $f'''(x) = 100$. Do you understand these results?
12. Produce a table of x_i and $e^{\sin(2x_i)}$ values, where x_i goes from 0 to 1.6 in steps of 0.08.

    (a) Plot the forward-difference and central-difference results (for the first derivative) given these values. (Hint: if you cannot produce a result for a specific x, don't.) Then, introduce a curve for the analytical derivative.
    (b) Use Richardson extrapolation for the forward difference for points on a grid and add an extra set of points to the plot. You can use:

$$R_{fd} = 2D_{fd}(h) - D_{fd}(2h) + O(h^2) \tag{3.91}$$

13. In the main text we used Richardson extrapolation in order to re-derive a second forward-difference formula for the first derivative, see Eq. (3.49). Do the same (i.e., don't use Taylor series) for the third forward difference for the first derivative.
14. In the main text we derived a second central-difference formula for the first derivative in two distinct ways, using Taylor series and using Richardson extrapolation. Correspondingly derive a second central-difference formula for the second derivative.
15. Explicitly show how automatic differentiation works on the following function:

$$f(x) = \frac{(x-5)(x-6)\sqrt{x}}{x-7} + \ln(8x) \tag{3.92}$$

    to evaluate $f(4)$ and $f'(4)$.
16. For this problem you will need to make sure JAX is installed on your system. Use the `grad()` function to automatically differentiate $f(x) = e^{\sin(2x)}$ at $x = 0.5$, comparing to the answer mentioned in the main text. Be sure to enable double precision.
17. Write a basic automatic differentiator in Python, implementing the operations discussed in section 3.4.1. Test your code on example functions like that in Eq. (3.58).
    Probably the easiest way you can accomplish this task is via object orientation: define a new `class` called `dual`; you should implement the special methods `__add__`, `__sub__`, and so on, according to the rules given in Eq. (3.56). If you don't want each of these to start with a test using `isinstance()`, you should have your `__init__` convert constants as per Eq. (3.57).
18. This problem studies the classical and quantum harmonic oscillators.

    (a) First, we study the classical oscillator. Take the solution $x = x_0 \sin(\omega t)$, which assumes that the particle is at $x = 0$ at $t = 0$. Calculate $P_c(x)$ by determining what fraction of the total time the particle will spend in an interval dx around x. In other words, use the relation $P_c(x)dx = dt/T$, where T is the period of oscillation.
    (b) Plot the quantum harmonic oscillator eigenfunctions (squared) for $n = 3, 10, 20, 150$ and compare with the classical solution(s).

19. Implement the three-dimensional harmonic oscillator, allowing different n in each Cartesian component. Test your new function by using `kinetic()` and `psiqho()` (i.e., the three-dimensional energy should be the sum of three one-dimensional energies).

20. Rewrite `psibox()` such that it applies to the three-dimensional particle in a box. Your wave vector should take the form $\mathbf{k} = 2\pi(n_x, n_y, n_z)/L$. In practice, you may wish to only input the "cardinal number" labelling the eigenstate. To do so, order the triples n_x, n_y, n_z according to the magnitude of the sum of their squares. Rewrite `psibox()` such that it takes in only the cardinal number (and the box size L) and evaluates the correct wave function. (You will probably want to first create another function which produces a dictionary mapping cardinal numbers to triples n_x, n_y, n_z.)
21. Rewrite `kinetic()` to use the second central-difference formula for the second derivative (from an earlier problem). Compare your answers to what you get in `kinetic.py` for both the harmonic-oscillator and the periodic-boundary cases.
22. This problem applies our function `kinetic()` to a physical setting that is different from what we encountered in our Project. Specifically, we wish to study a particle in one dimension, impinging on a simple-step barrier, i.e., our potential is:

$$V(x) = \begin{cases} 0, & x < 0 \\ V_0, & x > 0 \end{cases} \tag{3.93}$$

and we take the energy of the particle to be $E < V_0$. As you may have seen in a course on quantum mechanics, the wave function for this problem takes the form:

$$\psi(x) = \begin{cases} Ae^{ikx} + Be^{-ikx}, & x < 0 \quad (k = \sqrt{2mE}/\hbar) \\ Ce^{-\kappa x}, & x > 0 \quad (\kappa = \sqrt{2m(V_0 - E)}/\hbar) \end{cases} \tag{3.94}$$

and continuity of the wave function and its derivative gives us:

$$\frac{C}{A} = \frac{2}{1 + i\kappa/k}, \qquad \frac{B}{A} = \frac{1 - i\kappa/k}{1 + i\kappa/k} \tag{3.95}$$

Take $k = 2$ and $\kappa = 4$ and determine (both analytically and using Python) what the kinetic energy should be if the particle is located at $x < 0$ (or at $x > 0$).