8 Differential Equations

One thing might find out how to learn many,
if we could get even a small beginning of hope.

Sophocles

8.1 Motivation

This is the final chapter in the book, so we will take advantage of several of the tools we have developed so far, most notably: finite differences (chapter 3), linear algebra (chapter 4), root-finding (chapter 5), discrete Fourier transforms (chapter 6), and quadrature rules (chapter 7). Our exposition is cumulative, so you are expected to have studied the preceding material; even so, we provide specific references when we use techniques introduced in earlier chapters, so you can brush up on selected topics if the need arises.

8.1.1 Examples from Physics

Differential equations are at the heart of physics and (together with linear algebra) are the most prominent part of numerical analysis. As usual, here are a few examples:

1. **Stellar structure with relativistic corrections**
   Assume you are given an *equation of state*, $P(\rho)$, relating the pressure P and the mass density ρ for a perfect fluid. We would like to see how to go from the equation of state (which describes microphysics) to the structure of a star (which is a macrophysical consequence). In other words, we wish to determine how the matter is distributed as you start from the center of a star and go all the way out to its edge. In general relativity, the structure of a spherically symmetric static star can be determined by combining the Einstein field equations with an equation describing hydrostatic equilibrium; this gives rise to the following relation, where c is the speed of light:

$$\frac{dP(r)}{dr} = -\frac{Gm(r)\rho(r)}{r^2}\left[1+\frac{P(r)}{\rho(r)c^2}\right]\left[1+\frac{4\pi r^3 P(r)}{m(r)c^2}\right]\left[1-\frac{2Gm(r)}{c^2 r}\right]^{-1} \tag{8.1}$$

   known as the *Tolman–Oppenheimer–Volkoff (TOV) equation*; one way of recovering the Newtonian limit is to take $c^2 \to \infty$. Here, G is Newton's gravitational constant, r is the radial coordinate, and $m(r)$ is the mass inside a sphere of radius r, which satisfies:

$$\frac{dm(r)}{dr} = 4\pi r^2 \rho(r) \tag{8.2}$$

Note that in the last two relations P, ρ, and m are functions of only r; as a result we are dealing with a pair of *ordinary* differential equations. From the equation of state, given $\rho(r)$ you can determine $P(\rho(r))$, meaning that Eq. (8.1) and Eq. (8.2) are two first-order differential equations for $\rho(r)$ and $m(r)$; crucially, these cannot be solved independently of each other, since both of them involve $m(r)$ and $\rho(r)$; we say that these are *simultaneous differential equations* (or *coupled* differential equations).

We start solving these simultaneous differential equations at the center of the star (where we impose $\rho(0) = \rho_c$ and $m(0) = 0$) and then integrate outward. The mass M and radius R of the star are determined by $P(R) = 0$ and $m(R) = M$, i.e., the star ends when the pressure vanishes, at which radius we get the star's entire mass. Mathematically, this is an *initial-value problem*, even though there's no time involved here: "initial" means that we know the starting values and are trying to determine the behavior elsewhere.

2. **Projectile motion with air resistance**

Consider a projectile of mass m in two dimensions. Its motion will obey Newton's second law, i.e., $\mathbf{F} = m\mathbf{a}$; recall that $\mathbf{a} = d^2\mathbf{r}/dt^2$, where $\mathbf{r}$'s components are x and y. In two dimensions we get a pair of second-order differential equations for $x(t)$ and $y(t)$. Our projectile will feel the force of gravity and also air resistance; at large speeds, the latter can be expressed as follows:

$$\mathbf{F}_d = -km\mathbf{v}^2 \frac{\mathbf{v}}{|\mathbf{v}|} \tag{8.3}$$

This force is opposite to the direction of motion. The k is a parameter containing what is known as the drag coefficient, though the details are not relevant here: what matters is that the magnitude of the force in Eq. (8.3) is proportional to the square of the velocity. Putting everything together, Newton's second law takes the form:

$$\begin{aligned} \frac{d^2x}{dt^2} &= -k\frac{dx}{dt}\sqrt{\left(\frac{dx}{dt}\right)^2 + \left(\frac{dy}{dt}\right)^2} \\ \frac{d^2y}{dt^2} &= -g - k\frac{dy}{dt}\sqrt{\left(\frac{dx}{dt}\right)^2 + \left(\frac{dy}{dt}\right)^2} \end{aligned} \tag{8.4}$$

where the acceleration due to gravity, g, is present only in the y direction. Note that we have $x = x(t)$ and $y = y(t)$, i.e., both x and y depend only on t, so these are second-order ordinary differential equations; even so, our problem consists of two *simultaneous* ordinary differential equations, since $x(t)$ and $y(t)$ appear in both equations.

As you may recall, to start solving a second-order differential equation you need two pieces of information, the starting value of the function and the starting value of its derivative. For the projectile problem, we are faced with two second-order differential equations, so our input information would have to be the four quantities $x(0)$, $v_x(0)$, $y(0)$, and $v_y(0)$, where $v_x = dx/dt$ and $v_y = dy/dt$. In other words, we need to know the starting positions and starting velocities of the projectile. So far, this is another example of an initial-value problem, like the one we encountered in solving the TOV equation. Here's a twist, though: what if, instead, you were given only $x(0)$, $v_x(0)$, $y(0)$, and $y(T)$? These are still four numbers, namely the starting positions, one component of the starting velocity, and one component of the final position. In physical terms: you shoot a

cannonball at a given point and with a given v_x and you know how much time passes before it hits the ground (T where $y(T) = 0$). This is no longer an initial-value problem, since we don't have all the starting information at our disposal; instead, it is a *boundary-value problem*, where we know a final piece of information, which we'll try to use to extract the missing initial value, in this example $v_y(0)$.

3. **Schrödinger equation(s)**
   We encountered the Schrödinger equation in section 4.5: at the time, we were studying spins, so we needed its formulation in terms of state vectors. Here, we examine the (simpler) scenario of a single spinless non-relativistic particle of mass m in one spatial dimension, x, similarly to section 3.5. The system is described by its *wave function* $\Psi(x,t)$ which satisfies the *time-dependent Schrödinger equation* in the position basis:

$$i\hbar\frac{\partial\Psi(x,t)}{\partial t} = \left[-\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2} + V(x,t)\right]\Psi(x,t) \tag{8.5}$$

   The right-hand side is made up of a *kinetic* energy term (the second derivative) and a *potential* energy term. Observe that Eq. (8.5) involves a second-order spatial derivative and a first-order time derivative.[1] As you can see, the time-dependent Schrödinger equation is a linear partial differential equation; it is a dynamical equation, allowing us to determine how the wave function changes in time (i.e., it is an *initial-value problem*, for a partial differental equation this time).
   For the special case where the potential is time independent, $V(x)$, Eq. (8.5) is *separable*, giving rise to a trivial time dependence:

$$\Psi(x,t) = \psi(x)e^{-iEt/\hbar} \tag{8.6}$$

   with $\psi(x)$ obeying the *time-independent Schrödinger equation* in the position basis:

$$\left[-\frac{\hbar^2}{2m}\frac{d^2}{dx^2} + V(x)\right]\psi(x) = E\psi(x) \tag{8.7}$$

   Here the energy E is a real number. Crucially, Eq. (8.7) is no longer a dynamical equation but an *eigenvalue problem*: this equation has acceptable solutions only for specific values of E. In one spatial dimension, the time-independent Schrödinger equation is an *ordinary* differential equation. We had already seen the Schrödinger equation as an eigenproblem in the case of interacting spins in section 4.5, but there we were naturally faced with matrices. In the present case, we have an eigenvalue equation even though we're not using the language of matrices (yet).

8.1.2 The Problems to Be Solved

For most of this chapter, we will be tackling *ordinary differential equations* (ODEs). The simplest case is when we are trying to solve a single differential equation for a single function, $y(x)$, where x is known as the *independent variable* and y is the *dependent variable*, namely the solution we are after.

One could start by studying equations involving, say, second-order derivatives of y. For

[1] This asymmetry is lifted when you introduce special relativity, giving rise to the *Klein–Gordon equation*.

reasons that will become clear in section 8.2.4 we, instead, focus on the more elementary problem of a first-order derivative. Thus, the first class of equation we will tackle in this chapter will be the following *initial-value problem* (IVP):

$$y'(x) = f\left(x, y(x)\right), \qquad y(a) = c \tag{8.8}$$

where $y' = dy/dx$ and $f(x, y)$ is a known function (in general nonlinear) which could depend on both x and y,[2] e.g., $f(x, y) = 3x^5 - y^3$. We emphasize that f is known: what we wish to solve for is y. This is called an *initial-value* problem because we know the value of $y(x)$ at a certain point, a (i.e., we know that $y(a) = c$, where c is given), so we start from there and try to integrate up to, say, the point b. We used the word "integrate", bringing to mind the theme of chapter 7. Formally, we can integrate Eq. (8.8) to find:

$$y(x) = c + \int_a^x f(z, y(z))dz \tag{8.9}$$

If we now pick the special case where f does not depend on y (i.e., the opposite of an autonomous ODE) and set $x = b$, then our problem looks similar to Eq. (7.7). Thus, you should not be surprised to find out that terms introduced in the previous chapter (midpoint, trapezoid, Simpson's) will re-appear in the present chapter. That being said, we are here interested in producing not a single number (the value of a definite integral) but a full function, $y(x)$; also, in practice the function f most often *does* depend on y, so the problem we are faced with is more complicated than that of the previous chapter.

For a higher-order ODE, we would have to specify all necessary information at the starting point; for example, in an initial-value problem for a second-order ODE we would need to specify the values of both y (the sought-after function) and y' (its derivative) at a:

$$y'' = f(x, y, y'), \qquad y(a) = c, \qquad y'(a) = d \tag{8.10}$$

In this, more general, case we see that f could also depend on y'; in other words, we have isolated the highest-order derivative on the left-hand side. Observe that both $y(a)$ and $y'(a)$ are given at the same point, so this is still an initial-value problem. A much harder problem, which we will also discuss in this chapter, arises when you are given the values of the function at two distinct points, without being provided the starting value of y'. This is known as a *boundary-value problem* (BVP), which we will tackle in its simplest possible form, namely that of a second-order ODE:

$$y'' = f(x, y, y'), \qquad y(a) = c, \qquad y(b) = d \tag{8.11}$$

Our input data, $y(a) = c$ and $y(b) = d$, are known as *boundary conditions*. While in Eq. (8.10) we always got a unique solution, BVPs are more complicated: they can have no solutions or infinitely many solutions, but we will mainly focus on BVPs which have a single solution.

[2] If f depends on x only through its dependence on $y(x)$, we are dealing with an *autonomous* ODE.

An even harder problem arises when f in Eq. (8.11) is generalized to depend not only on x, y, and y', but also on a parameter s; this is an *eigenvalue problem* (EVP):

$$y'' = f(x, y, y'; s), \qquad y(a) = c, \qquad y(b) = d \tag{8.12}$$

This equation only has non-trivial solutions for special values of s. This means that we will need to computationally find the appropriate values of s. Viewed from another perspective, in Eq. (8.10) or Eq. (8.11) we have a single equation which we try to solve, but our newest problem in Eq. (8.12) describes a family of equations (and correspondingly more than one solutions). Unsurprisingly, such an eigenvalue problem has close connections both to the matrix eigenvalues which we encountered in chapter 4 and to the eigenvalues appearing in the Schrödinger equation, as touched upon in section 8.1.1 (where the role played by s here was played by the eigenenergy E).

All three problems discussed above appeared in the context of *ordinary differential equations*. Of course, in practice one is also faced with *partial differential equations* (PDEs), namely equations where the dependent variable depends on more than one independent variable. Here we will provide only the briefest of introductions to the subject, in section 8.5. Specifically, we will tackle *Poisson's equation* in two dimensions:

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = f(x, y) \tag{8.13}$$

employing a method that is different from what we used for ODEs (though tools that were introduced earlier in the chapter are certainly also applicable to PDEs). Note that here our dependent variable is ϕ, which depends on both x and y, i.e., $\phi(x, y)$. Crucially, this y is *not* a dependent variable as in previous paragraphs.[3]

8.2 Initial-Value Problems

While this chapter discusses a variety of questions, most of it is dedicated to initial-value problems, since the techniques that can be used to tackle these have wider applicability and will re-appear later on. As mentioned above, we will write a general IVP in the form:

$$y'(x) = f\left(x, y(x)\right), \qquad y(a) = c \tag{8.14}$$

where $y' = dy/dx$ and $f(x, y)$ is known. We wish to solve this equation for $y(x)$, with x going from a to b.

[3] You may wish to think of this problem as $y(x_0, x_1)$ instead, but it's important not to get confused: x_0 and x_1 would then be independent variables in their own right, not points (e.g., x_0 could be varied from a to b).

In most of this chapter, we will employ a *discretization*, more specifically an equally spaced grid of points, as for Newton–Cotes quadrature, Eq. (7.8). In other words, we will be trying to compute the function $y(x)$ at a set of n grid points x_j, from a to b:

$$x_j = a + jh \tag{8.15}$$

where, as usual, $j = 0, 1, \ldots, n-1$. The step size h is obviously given by:

$$h = \frac{b-a}{n-1} \tag{8.16}$$

We could use different step sizes h in different regions, as touched upon in section 8.2.3, but we will mostly stick to a fixed h, as per Eq. (8.16), in what follows.

Let us introduce some more notation. At a given grid point, x_j, the exact solution of our ODE will be $y(x_j)$; we will use the symbol y_j to denote the approximate solution, i.e., the value that results from a given discretization scheme. We now turn to a discussion of specific methods that will help us make y_j increasingly closer to $y(x_j)$.

8.2.1 Euler's Method

The simplest possible approach is known as *Euler's method*. In what follows, we will motivate it, discuss its error behavior, as well a scenario where it gets in trouble (and then we'll see what we can do about that).

Forward Euler

We can motivate what is known as the *forward Euler method* starting from the forward-difference formula. Translating Eq. (3.8) into our present notation gives:

$$y'(x_j) = \frac{y(x_{j+1}) - y(x_j)}{h} - \frac{h}{2}y''(\xi_j) \tag{8.17}$$

where, as usual, ξ_j is a point between x_j and x_{j+1}. Here it is assumed that we are starting at x_j and trying to figure out how to make a step onto the next point, x_{j+1}. Now, we know that $y(x_j)$ is the exact solution at the point x_j, so it must satisfy our ODE, Eq. (8.14):

$$y'(x_j) = f(x_j, y(x_j)) \tag{8.18}$$

We can use this equation to eliminate the first derivative from the left-hand side of Eq. (8.17):

$$y(x_{j+1}) = y(x_j) + hf(x_j, y(x_j)) + \frac{h^2}{2}y''(\xi_j) \tag{8.19}$$

We've made no approximations, which is why we don't see any y's with indices.

It is now time to make an approximation. Assuming h is small, the term proportional to h^2 will be less important, so we can drop it. This leads to the following prescription:

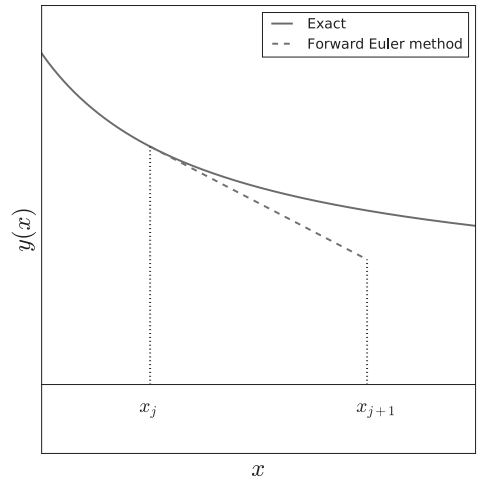


Fig. 8.1 Illustration of a step of the forward Euler method

$$\begin{aligned} y_{j+1} &= y_j + hf(x_j, y_j), \qquad j = 0, 1, \ldots, n-2 \\ y_0 &= c \end{aligned} \tag{8.20}$$

Since we started from the forward-difference formula in Eq. (8.17), the resulting method is known as the *forward* Euler method. Note that this is an approximate formula, involving y_j and y_{j+1} instead of $y(x_j)$ and $y(x_{j+1})$. As you have seen, higher-order terms in the Taylor expansion are ignored. We'll see how to do (much) better in later sections.

The method is illustrated in Fig. 8.1: since the right-hand side of Eq. (8.20) contains a term proportional to $f(x_j, y_j)$, and we know from our starting equation, Eq. (8.14), that $f(x_j, y_j)$ is (an approximation to) the slope of the tangent to the exact solution at x_j, the geometrical interpretation is reasonably straightforward. Note that in this figure we assumed that $y(x_j) = y_j$ for the purposes of illustration. This is certainly true when we start (i.e., $y(x_0) = y_0$), but will most likely not continue to be true later on (i.e., the starting point of the dashed line will not lie on top of the exact solution). Similarly, the forward Euler method approximates the slope as $f(x_j, y_j)$, whereas the true slope would have been $f(x_j, y(x_j))$. However, that's inevitable: at a given iteration, we have only y_j at our disposal, i.e., our approximation to the true $y(x_j)$.

Local and Global Error

We turn to the error incurred in a single step of the forward Euler method. This is known as the *local truncation error*; you can think of this as the error made when you go from x_0 to x_1, i.e., the difference between $y(x_1)$ and y_1. More generally, we define:

$$t_j = y(x_{j+1}) - y(x_j) - hf(x_j, y(x_j)) \tag{8.21}$$

This is defined in terms of exact quantities ($y(x_j)$, not y_j): this is because we are interested in the error incurred in a single step *assuming* we were starting from the exact solution. For the forward Euler method, we can immediately plug Eq. (8.19) into Eq. (8.21) to find:

$$t_j = \frac{1}{2}h^2 y''(\xi_j) \tag{8.22}$$

Thus, the local discretization error of the forward Euler method is $O(h^2)$. This is reminiscent of the absolute error in the one-panel rectangle formula, as per Eq. (7.17).[4]

Despite the simplicity (and popularity) of the analogy with the rectangle rule, if you wanted to proceed by analogy to the *composite* rule, as per Eq. (7.18), you would be on shaky ground. The reason is that here the different steps are *not* independent from one another: the actual error at the j-th step is *not* given by Eq. (8.22) because, as hinted at above, $y(x_j)$ is actually different from y_j and therefore $f(x_j, y(x_j))$ is also different from $f(x_j, y_j)$, due to the errors made in previous iterations; in other words, the local truncation error doesn't tell the whole story.

Let's be clear about our goal: we wish to determine the *global error* that accumulates when we numerically integrate our differential equation from $x_0 = a$ to $x_{n-1} = b$, i.e.,

$$\mathcal{E} = y(b) - y_{n-1} \tag{8.23}$$

As in the previous chapter, this definition is of the form "exact minus approximate". Working toward this goal, let us introduce some new notation:

$$e_j = y(x_j) - y_j \tag{8.24}$$

where, obviously, $e_{n-1} = \mathcal{E}$. We are consciously not calling this new quantity $\mathcal{E}_j$, in order to emphasize the conceptual differences from what was carried out in the previous chapter. With a view to seeing how the error in one step is related to that in the next step, subtract Eq. (8.20) from Eq. (8.19) to find:

$$e_{j+1} = e_j + h\Big[f(x_j, y(x_j)) - f(x_j, y_j)\Big] + t_j \tag{8.25}$$

where we also made use of Eq. (8.22). If we take the absolute value and use the triangle inequality, this gives:

$$|e_{j+1}| \le |e_j| + h\Big|\Big[f(x_j, y(x_j)) - f(x_j, y_j)\Big]\Big| + |t_j| \tag{8.26}$$

If we now assume that f satisfies Lipschitz continuity[5] and also that the second derivatives (contained in t_j) are bounded by M, we get:

$$|e_{j+1}| \le (1 + Lh)|e_j| + \frac{Mh^2}{2} \tag{8.27}$$

where we grouped together the first two terms on the right-hand side.

[4] Some authors define the local truncation error with a factor of h divided out. We work by analogy with the scaling in chapter 7, where the local error is always one order higher than the global error.

[5] Lipschitz continuity means that there exists a constant L such that $|f(x, y) - f(x, \bar{y})| \le L\,|y - \bar{y}|$ holds. This is satisfied if $f(x, y)$ has bounded partial derivatives, as will be the case for us.

For $j = 0$, Eq. (8.27) gives:

$$|e_1| \leq \frac{Mh^2}{2} \tag{8.28}$$

since $e_0 = y(x_0) - y_0 = 0$. Similarly, for $j = 1$ we find:

$$|e_2| \leq (1 + Lh)|e_1| + \frac{Mh^2}{2} \leq [1 + (1 + Lh)]\,\frac{Mh^2}{2} \tag{8.29}$$

and the next step gives:

$$|e_3| \leq (1 + Lh)|e_2| + \frac{Mh^2}{2} \leq \left[1 + (1 + Lh) + (1 + Lh)^2\right]\frac{Mh^2}{2} \tag{8.30}$$

Repeating this until the end, and summing up the geometric series, leads to:

$$|\mathcal{E}| = |e_{n-1}| \leq \frac{(1 + Lh)^{n-1} - 1}{Lh}\,\frac{Mh^2}{2} \tag{8.31}$$

where we can cancel the h in the denominator.

A Maclaurin expansion for the exponential, as per Eq. (C.2), gives $e^x = 1 + x + x^2 e^{\xi}/2$. This implies that $e^x \geq 1 + x$ and therefore $(1 + x)^{n-1} \leq e^{(n-1)x}$. Applied to our result:

$$|\mathcal{E}| \leq h\frac{M}{2L}(e^{L(b-a)} - 1) \tag{8.32}$$

where we also used $(n - 1)h = b - a$. Even if you need to re-read the above derivation, the main conclusion is clear: the forward Euler method converges and its global error decreases linearly, $O(h)$. While a $O(h)$ scaling is not great, it does imply that as h goes to 0 the global error also goes to 0 (ignoring considerations arising from roundoff error). We say that this is a *convergent* method. To reiterate, up to this point we were interested in what happened as we kept making h smaller; we now turn to what happens for a fixed h (and find out that we can get into serious trouble).

Stability Considerations

We now go over a pathological scenario: a case where the forward Euler method simply fails to converge. Put another way, we'll see an example where a small change in the step size has a huge effect on the global error, considerably beyond what is captured in our $O(h)$ result.

To make the discussion concrete, we will examine the (aptly named) *test equation*:

$$y'(x) = \mu\, y(x), \qquad y(0) = 1 \tag{8.33}$$

where μ is real (and constant). This is a very simple autonomous ordinary differential equation. Its exact solution can be straighforwardly seen to be $y(x) = e^{\mu x}$.

The steps to follow will be for the test equation only but, as you will find out in the

problem set, an appropriately generalized derivation can be carried out for any differential equation. Basically, the behavior of solutions to the test equation is pretty representative of more general features, so this equation will re-appear later in the chapter.

Stability of Forward Euler Method

Applying the forward Euler method, Eq. (8.20), to the test equation, Eq. (8.33), leads to $y_{j+1} = y_j + h\mu y_j$, which can be re-expressed in the form:

$$y_{j+1} = (1 + \mu h) y_j \tag{8.34}$$

If we now repeatedly apply this equation from $j = 0$ to $j = n - 2$, we get:

$$y_{n-1} = (1 + \mu h)^{n-1} y_0 = (1 + \mu h)^{n-1} \tag{8.35}$$

where, in the last step, we took advantage of the fact that $y_0 = 1$.

When $\mu > 0$, our result in Eq. (8.35) grows pretty rapidly, but that's OK, since so does the exact solution $y(x) = e^{\mu x}$. We are more interested in the case of $\mu < 0$, for which the exact solution is decaying exponentially. We immediately realize that $(1 + \mu h)^{n-1}$ may or may not decay as n gets larger, depending on the value of μh. Thus, if we wish to ensure that the solution gets smaller from one step to the next, i.e., $|y_{j+1}| < |y_j|$, we see from Eq. (8.34) that we need to demand that:

$$|1 + \mu h| < 1 \tag{8.36}$$

holds. In other words, we have arrived at the following *stability condition*:

$$h < \frac{2}{|\mu|} \tag{8.37}$$

If this is not satisfied, the forward Euler method leads to a numerically unstable solution.

In Fig. 8.2 we show the result of applying the forward Euler method to the test equation $y'(x) = -20y(x)$, i.e., for the case of $\mu = -20$. Based on the above criterion, we expect that a qualitatively new feature emerges when we cross the $h = 0.1$ threshold. This expectation is clearly borne out by the figure. The global error of the forward Euler method is supposed to be $O(h)$, but when we change $n = 12$ to $n = 10$ (i.e., go from $h = 0.091$ to $h = 0.111$) we encounter a dramatic change in behavior. As a matter of fact, the $n = 10$ results oscillate and grow considerably away from the exact solution; clearly, for this step size and μ, the forward Euler method is *unstable*. To reiterate the main point: while accuracy requirements may lead us to expect a given behavior for a given step size, stability requirements have to be taken into account separately. In other words, the forward Euler global error is $O(h)$, *only when the method is stable*. This behavior is rarely acceptable, so we now try to see if we can come up with other techniques that perform better.

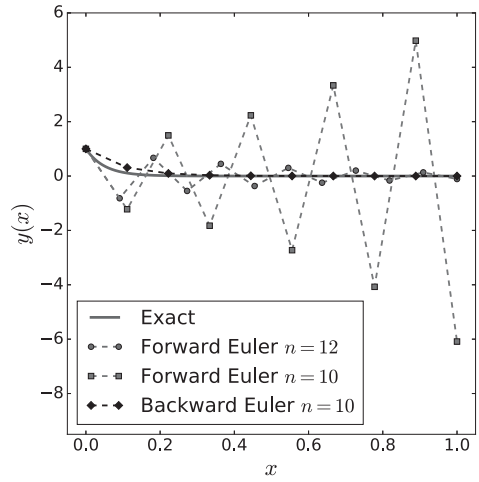


Fig. 8.2 The forward and backward Euler methods applied to the test equation

Backward Euler Method

In Fig. 8.2 we also show a set of results for a mysterious "backward Euler" method, so let us now see what that is. We will introduce this technique at a general level, i.e., for any (first-order) differential equation. After that, we will see how the new method helps us do better for the problem of the test equation.

The forward Euler method is so named because we motivated it starting from the forward-difference approximation to the first derivative. We can do something analogous for the backward-difference approximation; this means we can translate Eq. (3.11) to find:

$$y'(x_{j+1}) = \frac{y(x_{j+1}) - y(x_j)}{h} + \frac{h}{2} y''(\xi_j) \tag{8.38}$$

This certainly seems like an innocuous variation on what we were doing earlier. However, we'll now see that its impact is major. Recall that we are trying to solve our general ODE, Eq. (8.14); since $y(x_{j+1})$ is the exact solution at the point x_{j+1}, we can eliminate the first derivative from the left-hand side of Eq. (8.38), getting:

$$y(x_{j+1}) = y(x_j) + hf(x_{j+1}, y(x_{j+1})) - \frac{h^2}{2} y''(\xi_j) \tag{8.39}$$

This motivates our approximation, which is to drop the term proportional to h^2:

$$\begin{aligned} y_{j+1} &= y_j + hf(x_{j+1}, y_{j+1}), \qquad j = 0, 1, \ldots, n-2 \\ y_0 &= c \end{aligned} \tag{8.40}$$

Since this time we started from the backward-difference formula in Eq. (8.38), the resulting method is known as the *backward* Euler method. We observe that the local truncation error

(i.e., the $O(h^2)$ term we dropped to go from Eq. (8.39) to Eq. (8.40)) is almost identical to what we were faced with in the forward Euler case, but has the opposite sign.

While Eq. (8.40) appears, at first sight, to be quite similar to Eq. (8.20), in reality it is very different: in Eq. (8.20) we have a way of producing the y_{j+1} on the left-hand side simply by plugging in results from earlier steps, such as y_j; methods that allow you to do this are called *explicit*. However, Eq. (8.40) involves y_{j+1} on both the left-hand side and the right-hand side. Since $f(x, y)$ is in general a nonlinear function, we are thereby faced with a root-finding problem of the form:

$$z = y_j + hf(x_{j+1}, z) \tag{8.41}$$

at each iteration! Methods for which the evaluation of y_{j+1} implicitly depends on y_{j+1} itself are called *implicit*. Based on what you know from chapter 5, in the general case such methods require considerably more computational effort per iteration than analogous explicit methods; this immediately raises the question why anyone would bother implementing them.

Stability of Backward Euler Method

To give the conclusion ahead of time: the backward Euler method has much better stability properties, thereby justifying the extra work we need to put in per iteration. This happens to be a general feature: *implicit methods are often better from the perspective of stability*. In the rest of the chapter we will introduce a mixture of explicit and implicit methods: the former are typically faster but, when they get in trouble, implicit methods can save the day.

In order to examine the stability features of the backward Euler method, we turn once again to the test equation of Eq. (8.33). For such a simple right-hand side, it is straightforward not only to write down Eq. (8.40):

$$y_{j+1} = y_j + h\mu y_{j+1} \tag{8.42}$$

but also to analytically solve it for y_{j+1}:

$$y_{j+1} = \frac{1}{1 - \mu h} y_j \tag{8.43}$$

As you go from the j-th step to the $(j + 1)$-th one, this equation shows you that your approximate solution for $\mu > 0$ may grow; however, that's fine, since we know that the exact solution also grows. In reality, we are more interested in the case of $\mu < 0$, for which the forward Euler method could get into serious trouble. In the present case, you can see from Eq. (8.43) that when $\mu < 0$, regardless of the magnitude of $h > 0$, you will always satisfy $|y_{j+1}| < |y_j|$. This is as it should be, since the exact solution will be decaying. In other words, we have shown that, for the test equation with $\mu < 0$, the backward Euler method is *unconditionally stable*, i.e., is stable regardless of the value of μh. We have thereby explained why this implicit method did so well in Fig. 8.2.

8.2.2 Second-Order Runge–Kutta Methods

At a big-picture level, the Euler method (whether explicit or implicit) followed from truncating a Taylor expansion to very low order. Thus, a way to produce increasingly better methods would be to keep more terms in the Taylor expansion. Of course, higher-degree terms are associated with derivatives of higher order, which are generally difficult or expensive to compute. In this and the following section, we will investigate an alternative route: so-called *Runge–Kutta methods* employ function evaluations (i.e., not derivatives) at x_j or x_{j+1} (or at points in between), appropriately combined such that the prescription's Taylor expansion matches the exact solution up to a given order. This may sound too abstract, so let's explicitly carry out a derivation that will lead to a family of second-order methods.

Derivation

Before considering the Runge–Kutta prescription (which will allow us to produce an approximate solution), let us examine the Taylor expansion of the exact solution. As advertised, we will explicitly evaluate higher-order terms this time. We have:

$$\begin{aligned} y(x_{j+1}) &= y(x_j) + hy'(x_j) + \frac{h^2}{2}y''(x_j) + \frac{h^3}{6}y'''(x_j) + O(h^4) \\ &= y(x_j) + hf(x_j, y(x_j)) + \frac{h^2}{2}f' + \frac{h^3}{6}f'' + O(h^4) \\ &= y(x_j) + hf(x_j, y(x_j)) + \frac{h^2}{2}\left(\frac{\partial f}{\partial x} + f\frac{\partial f}{\partial y}\right) \\ &\quad + \frac{h^3}{6}\left[\frac{\partial^2 f}{\partial x^2} + 2f\frac{\partial^2 f}{\partial x \partial y} + \frac{\partial f}{\partial x}\frac{\partial f}{\partial y} + f^2\frac{\partial^2 f}{\partial y^2} + f\left(\frac{\partial f}{\partial y}\right)^2\right] + O(h^4) \end{aligned} \tag{8.44}$$

In the first equality we expanded $y(x_j + h)$ around the point x_j. In the second equality we employed the fact that $y(x_j)$ is the exact solution at the point x_j, so it must satisfy our ODE, Eq. (8.14), namely $y'(x_j) = f(x_j, y(x_j))$; since the notation gets messy, we write f', where it is implied that the derivatives are taken at the point $x_j, y(x_j)$. To get to the third equality, we noticed that when evaluating f' we need to account for both the explicit dependence on x and for the fact that y depends on x:

$$f' = \frac{\partial f}{\partial x} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial x} = \frac{\partial f}{\partial x} + \frac{\partial f}{\partial y}f \tag{8.45}$$

Similarly, the next order term in the third equality applied this fact again:

$$f'' = \frac{\partial}{\partial x}\left(\frac{\partial f}{\partial x} + \frac{\partial f}{\partial y}f\right) + \frac{\partial}{\partial y}\left(\frac{\partial f}{\partial x} + \frac{\partial f}{\partial y}f\right)\frac{\partial y}{\partial x} \tag{8.46}$$

Our final result in Eq. (8.44) is an expansion of the exact solution, explicitly listing all the terms up to order h^3. Any Ansatz, such as those we will introduce below, will have to match these terms order by order, if it is to be correct up to a given order.

We now turn to the *second-order Runge–Kutta* prescription; as its name implies, this will turn out to match Eq. (8.44) up to order h^2 (i.e., it will have a local truncation error of $O(h^3)$). Crucially, it will accomplish this without needing to evaluate any derivatives:

$$y_{j+1} = y_j + c_0 h f(x_j, y_j) + c_1 h f\left[x_j + c_2 h, y_j + c_2 h f(x_j, y_j)\right] \tag{8.47}$$

The c_0, c_1, and c_2 will be determined below. This prescription requires two function evaluations in order to carry out a single step: it may look as if there are three function evaluations, but one of them is re-used. To see this, take:

$$\begin{aligned} k_0 &= h f(x_j, y_j) \\ y_{j+1} &= y_j + c_0 k_0 + c_1 h f\left[x_j + c_2 h, y_j + c_2 k_0\right] \end{aligned} \tag{8.48}$$

As advertised, this doesn't involve any derivatives; it *does* involve the evaluation of f at a point other than x_j, y_j: this involves a shifting of both x_j (to $x_j + c_2 h$), as well as a more complicated shifting of y_j (to $y_j + c_2 h f(x_j, y_j)$). However, if c_0, c_1, c_2, x_j, y_j, and f are all known, then the right-hand side can be immediately evaluated; in other words, we are here dealing with an *explicit* Runge–Kutta prescription.

Our plan of action should now be clear: expand the last term in Eq. (8.47), grouping the contributions according to which power of h they are multiplying; then, compare with the exact solution expansion in Eq. (8.44) and see what you can match:

$$\begin{aligned} f\left[x_j + c_2 h, y_j + c_2 h f(x_j, y_j)\right] = f(x_j, y_j) &+ c_2 h \frac{\partial f}{\partial x} \\ &+ c_2 h f(x_j, y_j) \frac{\partial f}{\partial y} + \frac{c_2^2 h^2}{2} \frac{\partial^2 f}{\partial x^2} + \frac{c_2^2 h^2}{2} 2 f(x_j, y_j) \frac{\partial^2 f}{\partial x \partial y} \\ &+ \frac{c_2^2 h^2}{2} f^2(x_j, y_j) \frac{\partial^2 f}{\partial y^2} + O(h^3) \end{aligned} \tag{8.49}$$

It may be worthwhile at this point to emphasize that in Eq. (8.44) we expanded $y(x_j + h)$, i.e., a function of a single variable. This is different from what we are faced with in the left-hand side of Eq. (8.49), namely f, which is a function of both our independent variable and our dependent variable. Our result on the right-hand side involves all the necessary partial derivatives, similarly to what we did in the last step of Eq. (8.44).

If we now plug Eq. (8.49) into Eq. (8.47) and collect terms, we find:

$$\begin{aligned} y_{j+1} = y_j + (c_0 + c_1) h f(x_j, y_j) &+ c_1 c_2 h^2 \left[\frac{\partial f}{\partial x} + f(x_j, y_j) \frac{\partial f}{\partial y}\right] \\ &+ \frac{c_1 c_2^2 h^3}{2} \left[\frac{\partial^2 f}{\partial x^2} + 2 f(x_j, y_j) \frac{\partial^2 f}{\partial x \partial y} + f^2(x_j, y_j) \frac{\partial^2 f}{\partial y^2}\right] + O(h^4) \end{aligned} \tag{8.50}$$

Comparing this result with the expansion for the exact solution, Eq. (8.44), we realize that we can match the terms proportional to h and to h^2 if we assume:

$$c_0 + c_1 = 1, \qquad c_1 c_2 = \frac{1}{2} \tag{8.51}$$

Unfortunately, this does not help us with the term that is proportional to h^3. Simply put,

the term in brackets in Eq. (8.44) contains more combinations of derivatives than the corresponding term in Eq. (8.50). Thus, since we have agreement up to order h^2, we have shown that the explicit Runge–Kutta prescription in Eq. (8.47) has a local truncation error which is $O(h^3)$, as desired.[6] Since the local error of our Runge–Kutta prescription is $O(h^3)$, you can guess that the global error is one order worse, i.e., $O(h^2)$, similarly to what we saw in chapter 7 for the midpoint and trapezoid quadrature rules. You can now see why the present section's heading is *second-order* Runge–Kutta.

Explicit Midpoint and Trapezoid Methods

There is more than one way to satisfy Eq. (8.51); here we will examine two possible choices. The first one obtains when you take the parameters to be:

$$c_0 = 0, \quad c_1 = 1, \quad c_2 = \frac{1}{2} \tag{8.52}$$

For this case, the prescription in Eq. (8.47) takes the form:

$$y_{j+1} = y_j + hf\left[x_j + \frac{h}{2}, y_j + \frac{h}{2} f(x_j, y_j)\right] \tag{8.53}$$

Things may be more transparent if we break up the evaluations into two stages:

$$\begin{aligned} k_0 &= hf(x_j, y_j) \\ y_{j+1} &= y_j + hf\left[x_j + \frac{h}{2}, y_j + \frac{k_0}{2}\right] \end{aligned} \tag{8.54}$$

This is an explicit method, since the right-hand side is already known; it is also a method that employs the midpoint between x_j and x_{j+1}. Observe that the second argument passed to f is $y_j + k_0/2$, namely the (approximation to the) slope of the tangent at the midpoint. As a result, this is known as the *explicit midpoint method*.

Its interpretation can be understood from Fig. 8.3: recall that Fig. 8.1 showed Euler's method employing the slope at the point x_j, y_j. In contradistinction to this, here we are finding the slope at the midpoint and then using that to make a move starting from the point x_j, y_j.[7] As a result, even in this illlustration it can be seen that the explicit midpoint method does a better job than Euler's method; of course, you already expected this to be the case, since you know the local (or global) error to be better by one order.

We now turn to our second choice of how to satisfy the conditions in Eq. (8.51):

$$c_0 = \frac{1}{2}, \quad c_1 = \frac{1}{2}, \quad c_2 = 1 \tag{8.55}$$

For this case, the prescription in Eq. (8.47) takes the form:

$$y_{j+1} = y_j + \frac{h}{2} f(x_j, y_j) + \frac{h}{2} f\left[x_j + h, y_j + hf(x_j, y_j)\right] \tag{8.56}$$

[6] This is better by one order compared to the local truncation error of the Euler method(s), which was $O(h^2)$. On the other hand, Euler needs a single function evaluation per step, whereas our new method requires two.

[7] As before, we are assuming here that $y_j = y(x_j)$ for the purposes of illustration.

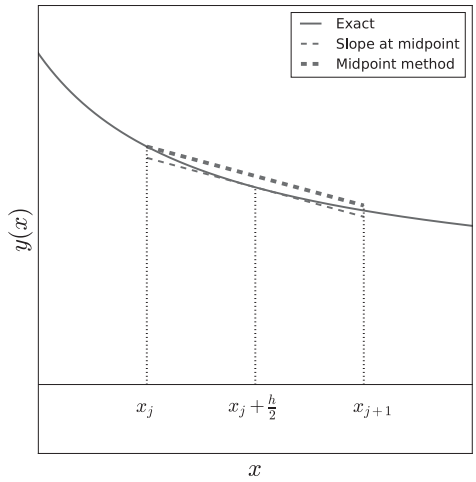


Illustration of a step of the midpoint method

Fig. 8.3

Once again, this is slightly easier to understand if we break up the evaluations in two stages:

$$\begin{aligned} k_0 &= hf(x_j, y_j) \\ y_{j+1} &= y_j + \frac{h}{2}\left[f(x_j, y_j) + f(x_{j+1}, y_j + k_0)\right] \end{aligned} \tag{8.57}$$

This, too, is an explicit method. Given that it involves a prefactor of $h/2$ multiplied with the sum of the function values at the left endpoint and (what appears to be) the right endpoint, you may not be surprised to hear that this is known as the *explicit trapezoid method*.[8] Just like the explicit midpoint method, this is a second-order Runge–Kutta technique; it requires two function evaluations per step and is better than the basic Euler method.

Quadrature Motivation

So far, we've mentioned terms from the previous chapter on numerical integration several times; we will now see in a bit more detail how these connections between the two chapters arise. In the process, we will also introduce two more (implicit) methods.

The goal, as before, is to figure out how to make the step from x_j to x_{j+1}, i.e., how to produce y_{j+1}. We have:

$$y(x_{j+1}) - y(x_j) = \int_{x_j}^{x_{j+1}} \frac{dy}{dx} dx = \int_{x_j}^{x_{j+1}} f(x, y(x)) dx \tag{8.58}$$

[8] These methods are also known by several other names. For example, the explicit trapezoid method is sometimes called *Heun's method* and the explicit midpoint method is known as the *modified Euler method*.

This is similar to what we did in Eq. (8.9), only this time we are limiting ourselves to a single step h. The first equality follows simply from the definition of a definite integral. The second equality has plugged in our ODE from Eq. (8.14), $y' = f(x, y)$. The different methods introduced earlier in this chapter will now be seen to be a result of evaluating the integral in the last step of Eq. (8.58) at varying levels of sophistication. Of course, it is important to note that the situation we are faced with here is not quite so clean: our f depends on both x and y (which in its turn also depends on x).

First, we approximate the integral using the left-hand rectangle rule of Eq. (7.10):

$$y_{j+1} = y_j + hf(x_j, y_j) \tag{8.59}$$

Notice how $y(x_j)$ turned into y_j and so on: we are making an approximation, in order to produce the next step in our method. Observe that this result is *identical* to the forward Euler method of Eq. (8.20). The connections between forward Euler and the rectangle rule aren't too surprising, since we've already seen that our expression for the local error in the two cases was fully analogous. Of course, we could have just as easily employed the right-hand rectangle rule, which would have led to:

$$y_{j+1} = y_j + hf(x_{j+1}, y_{j+1}) \tag{8.60}$$

Again, this is identical to the backward Euler method of Eq. (8.40). Of the last two methods, the first was explicit and the second implicit, as you know.

Now, we approximate the integral in Eq. (8.58) using the midpoint rule of Eq. (7.20):

$$y_{j+1} = y_j + hf\left[x_j + \frac{h}{2}, y\left(x_j + \frac{h}{2}\right)\right] \tag{8.61}$$

We are now faced with a problem: the last term involves $y(x_j + h/2)$ but that is off-grid. We don't (and will never) know the value of our dependent variable outside our grid points.[9] What we can do is to further approximate $y(x_j + h/2)$ by the average of y_j and y_{j+1}:

$$y_{j+1} = y_j + hf\left(x_j + \frac{h}{2}, \frac{y_j + y_{j+1}}{2}\right) \tag{8.62}$$

This is easily identifiable as an implicit method: it involves y_{j+1} on both the left-hand side and the right-hand side. (Note, however, that our earlier problem is resolved: y_j and y_{j+1} are approximate values *at* the grid points.) The resulting prescription is known as the *implicit midpoint method*. At this point, you could choose to approximate y_{j+1} on the right-hand side by the forward Euler method; if you do that, you will find that you recover the *explicit* midpoint method. If in the previous section you had any lingering doubts about why this method was so named, these should have been put to rest by now.

We can also approximate the integral in Eq. (8.58) using the trapezoid rule of Eq. (7.30):

[9] Of course, you could imagine changing the definition of your grid: if you take $h \to h/2$, then what used to be a midpoint is now a grid point. If you do that, you also have to update the equations appropriately.

$$y_{j+1} = y_j + \frac{h}{2}\left[f(x_j, y_j) + f(x_{j+1}, y_{j+1})\right] \tag{8.63}$$

You're probably getting the hang of it by now: since this is an implicit method that resulted from using the trapezoid rule, it is called the *implicit trapezoid method*. Just like we did in the previous paragraph, we could now approximate y_{j+1} on the right-hand side by the forward Euler method; unsurprisingly, doing that leads to the *explicit* trapezoid method. You may have noticed that we didn't actually prove that the *implicit* midpoint and trapezoid methods are second-order; the previous section carried out the Taylor-expansion matching for the *explicit* Runge–Kutta prescription in Eq. (8.47). One of the problems asks you to check the order of the error for the implicit case.

8.2.3 Fourth-Order Runge–Kutta Method

Up to this point, we've introduced six distinct methods: Euler, midpoint, and trapezoid, with each one appearing in an implicit or explicit version. We also saw the connections of these techniques with the corresponding quadrature rules from the previous chapter. Higher-order methods can be arrived at by applying higher-order quadrature rules; unfortunately, the connection betwen the former and the latter stops being as clean. One (unpleasant) option would then be to completely drop the connection with quadrature and revert to a Taylor expansion of a higher-order prescription, generalizing Eq. (8.47), and then matching terms order-by-order. One of the problems guides you toward that goal.

Before we go any farther, let us write down a very important prescription, belonging to the *fourth-order Runge–Kutta method*:

$$\begin{aligned}
k_0 &= hf(x_j, y_j) \\
k_1 &= hf\left(x_j + \frac{h}{2}, y_j + \frac{k_0}{2}\right) \\
k_2 &= hf\left(x_j + \frac{h}{2}, y_j + \frac{k_1}{2}\right) \\
k_3 &= hf\left(x_j + h, y_j + k_2\right) \\
y_{j+1} &= y_j + \frac{1}{6}\left(k_0 + 2k_1 + 2k_2 + k_3\right)
\end{aligned} \tag{8.64}$$

This Ansatz is sufficiently widespread that it is also known as *classic Runge–Kutta* or *RK4*. It requires four function evaluations in order to produce y_{j+1} starting from y_j; the fact that it can accomplish this task tells us that this is an explicit method.

Just like our earlier prescriptions, RK4 does *not* need to evaluate any derivatives. Even though it is more costly than any of the other methods we've seen up to now, it is certainly a worthwhile investment, as RK4 has a local error of $O(h^5)$, which is two orders better than

anything we encountered before. There are many other methods on the market, e.g., higher-order Runge–Kutta techniques, multistep methods, Richardson-extrapolation approaches, or techniques tailored to specific types of ODEs; the problem set introduces several of these. For most practical purposes, however, the fourth-order Runge–Kutta method is an efficient and dependable tool which is likely to be your technique of choice.

The broad outlines of a graphic interpretation of Eq. (8.64) are easy to grasp: the prescription involves evaluations at x_j and x_{j+1}, as well as at their midpoint. Similarly, k_0, k_1, k_2, and k_3 are approximations to the slope at the endpoints and the midpoint. Of course, you may be wondering exactly why these k_i's have been chosen to have this specific form (and also end up being combined in the final line in precisely that manner). As mentioned above, a full Taylor-expansion derivation is quite messy, without providing any new insights. However, the essential points can be grasped if we examine two extreme cases for $f(x, y)$: (a) the case where f does not depend on y, and (b) the case where f does not depend on x (an autonomous ODE). We take these up in turn.

First Case: Quadrature

As advertised, we first consider the case where f depends only on our independent variable x. Then, Eq. (8.58) takes the form:

$$y(x_{j+1}) - y(x_j) = \int_{x_j}^{x_{j+1}} f(x)dx \tag{8.65}$$

This is now not *similar* to what we had in the previous chapter, but *identical*. We've already seen the rectangle, midpoint, and trapezoid rule approximations to this integral. The next quadrature formula we encountered in the previous chapter was *Simpson's rule*, see Eq. (7.45). Applied to Eq. (8.65), this gives:

$$y_{j+1} = y_j + \frac{h}{6}\left[f(x_j) + 4f\left(x_j + \frac{h}{2}\right) + f(x_{j+1})\right] \tag{8.66}$$

where we took care of the fact that Eq. (7.45) was written down for two panels (i.e., from x_j to x_{j+2}). If we now look at the RK4 prescription of Eq. (8.64), we realize that when f does not depend on y we have $k_1 = k_2 = hf(x_j + h/2)$. Thus, Eq. (8.66) exactly matches the RK4 prescription. We have therefore accomplished (the first part of) what we set out to do: the fourth-order Runge–Kutta method is equivalent to Simpson's rule (for the, simpler, case where f does not depend on y). This is one way of seeing that RK4 has a local error of $O(h^5)$, as shown for Simpson's rule in Eq. (7.52).

Incidentally, the connections between the numerical integration of ODEs, on the one hand, and plain quadrature, on the other, do not stop here. We've focused on Newton–Cotes methods, but one could just as well employ Gauss–Legendre quadrature schemes; as a matter of fact, the midpoint method is a trivial example of this. One could, analogously, introduce fourth-order or sixth-order implicit Gauss–Legendre Runge–Kutta methods. That being said, we won't need to do so, since RK4 is enough for our purposes.

Second Case: Autonomous

So far, we've only accomplished half of what we wanted: we've seen what happens when we are faced with $f(x)$. We'll now examine the other extreme, that of an autonomous differential equation, namely one with $f(y)$. To make things concrete, we will study (again) the test equation of Eq. (8.33):

$$y'(x) = \mu y(x) \tag{8.67}$$

Since this is a case where we explicitly know the right-hand side (which also happens to be simple), we can follow the explicit route we opted against in the general case: we can compare the RK4 prescription with a Taylor expansion of the exact solution.

We start by explicitly applying the RK4 prescription of Eq. (8.64) to the test equation:

$$\begin{aligned}
k_0 &= hf(x_j, y_j) = \mu h y_j \\
k_1 &= hf\left(x_j + \frac{h}{2}, y_j + \frac{k_0}{2}\right) = \left[\mu h + \frac{(\mu h)^2}{2}\right] y_j \\
k_2 &= hf\left(x_j + \frac{h}{2}, y_j + \frac{k_1}{2}\right) = \left[\mu h + \frac{(\mu h)^2}{2} + \frac{(\mu h)^3}{4}\right] y_j \\
k_3 &= hf\left(x_j + h, y_j + k_2\right) = \left[\mu h + (\mu h)^2 + \frac{(\mu h)^3}{2} + \frac{(\mu h)^4}{4}\right] y_j \\
y_{j+1} &= y_j + \frac{1}{6}\left(k_0 + 2k_1 + 2k_2 + k_3\right) = \left[1 + \mu h + \frac{(\mu h)^2}{2} + \frac{(\mu h)^3}{6} + \frac{(\mu h)^4}{24}\right] y_j
\end{aligned} \tag{8.68}$$

The last step in the first four lines plugs in $f(y) = \mu y$ (and earlier results as they are produced). The last step in the final line grouped all the terms together.

We now recall that the exact solution to the test equation is $y(x) = ce^{\mu x}$. We can apply this at the two grid points of interest:

$$y(x_j) = ce^{\mu x_j}, \qquad y(x_{j+1}) = ce^{\mu(x_j+h)}, \qquad \frac{y(x_{j+1})}{y(x_j)} = e^{\mu h} \tag{8.69}$$

where in the last equation we formed the ratio, in order to emphasize that the specific starting point does not matter here. As usual, we now Taylor expand $e^{\mu h}$ up to a sufficienty high order:

$$\frac{y(x_{j+1})}{y(x_j)} = 1 + \mu h + \frac{(\mu h)^2}{2} + \frac{(\mu h)^3}{6} + \frac{(\mu h)^4}{24} + \frac{(\mu h)^5}{120} + O(h^6) \tag{8.70}$$

Comparing y_{j+1}/y_j to $y(x_{j+1})/y(x_j)$ we find perfect agreement up to h^4; significantly, when evaluating y_{j+1}/y_j we did *not* have to evaluate any derivatives! The RK4 prescription does not capture the term $(\mu h)^5/120$ (or any higher-order terms), so we see that the local error is $O(h^5)$. This completes the second half of our investigation of the local error of RK4. Since this is $O(h^5)$, we see that the global error of the fourth-order Runge–Kutta prescription of Eq. (8.64) is $O(h^4)$, as expected.

Code 8.1 ivp_one.py

```
import numpy as np

def f(x,y):
     return - (30/(1-x**2)) + ((2*x)/(1-x**2))*y - y**2

def euler(f,a,b,n,yinit):
     h = (b-a)/(n-1)
     xs = a + np.arange(n)*h
     ys = np.zeros(n)
     y = yinit
     for j,x in enumerate(xs):
          ys[j] = y
          y += h*f(x, y)
     return xs, ys

def rk4(f,a,b,n,yinit):
     h = (b-a)/(n-1)
     xs = a + np.arange(n)*h
     ys = np.zeros(n)
     y = yinit
     for j,x in enumerate(xs):
          ys[j] = y
          k0 = h*f(x, y)
          k1 = h*f(x+h/2, y+k0/2)
          k2 = h*f(x+h/2, y+k1/2)
          k3 = h*f(x+h, y+k2)
          y += (k0 + 2*k1 + 2*k2 + k3)/6
     return xs, ys

if __name__ == '__main__':
     a, b, n, yinit = 0.05, 0.49, 12, 19.53
     xs, ys = euler(f,a,b,n,yinit); print(ys)
     xs, ys = rk4(f,a,b,n,yinit); print(ys)
```

Implementation

While developing all these ODE integration techniques up to this point, we didn't say anything about whether or not f is linear or nonlinear. This is because for the initial-value

problem of an ordinary differential equation our methods are so robust that the nonlinearity of the equation typically doesn't really matter.[10] To drive this point home we will now solve the following (mildly nonlinear) *Riccati equation*:

$$y'(x) = -\frac{30}{1-x^2} + \frac{2x}{1-x^2}y(x) - y^2(x), \qquad y(0.05) = 19.53 \tag{8.71}$$

The right-hand side here depends on both x and y; recall that the nonlinearity comes from $y^2(x)$, not from the x dependence. Since this is an initial-value problem, we have pulled an initial value out of a hat, in order to get things going.

We are now ready to solve this equation in Python, using two of the seven numerical-integration techniques introduced so far; the result is Code 8.1. If you've been reading this book straight through, your programming maturity will have been developing apace. Thus, our comments on this and the following codes will be less extensive than in earlier chapters.

We start with a function implementing the right-hand side of Eq. (8.71). As usual, the main workhorses of the code are general, i.e., they allow you to solve a different ODE with a different initial value with minimal code modifications. This is obvious in the definition of `euler()`, which implements the forward Euler method of Eq. (8.20) and takes in as parameters the right-hand side of the ODE to be solved, the starting and ending points, the number of integration points to be used, as well as the initial value (i.e., the value of the dependent variable at the starting point). The body of the function is standard: we define the step size, set up the grid of x_j's, initialize y_0, and then step through the y_j's as per Eq. (8.20). We try to be Pythonic in the sense of minimizing index use, but even so we require a call to `enumerate()` to help us store the latest y_j in the appropriate slot. The function `rk4()` is a straightforward generalization of `euler()`: this time we are implementing Eq. (8.64) line by line. The main program simply sets up the input parameters and calls the two solvers one after the other.

In Fig. 8.4 we have taken the liberty of plotting the output produced in the main program. We've also included a curve marked as the exact solution: we won't tell you how we produced that for now (but stay tuned). Overall, we see that the fourth-order Runge–Kutta method does an excellent job integrating out from our starting point, despite the fact that we used a very coarse grid, i.e., only 12 points in total. The same cannot quite be said about the forward Euler method: already after the first point, we are visibly distinct from the exact solution. As a matter of fact, for this specific ODE, it is the first and last steps for which Euler does a bad job: these are the regions where the function is most interesting. Since Euler doesn't build in extra information (other than the slope at the left point), it fails to capture complicated behavior. In a problem, you will implement the explicit midpoint and trapezoid methods for the same ODE in order to see if they do any better.

[10] It may appear that this statement is somewhat softened in the following subsection, but even there the cause of trouble is not the nonlinearity.

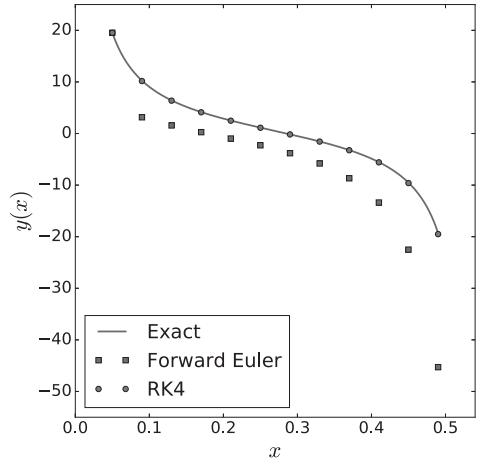


Fig. 8.4 Forward Euler and Runge–Kutta methods for our nonlinear initial-value problem

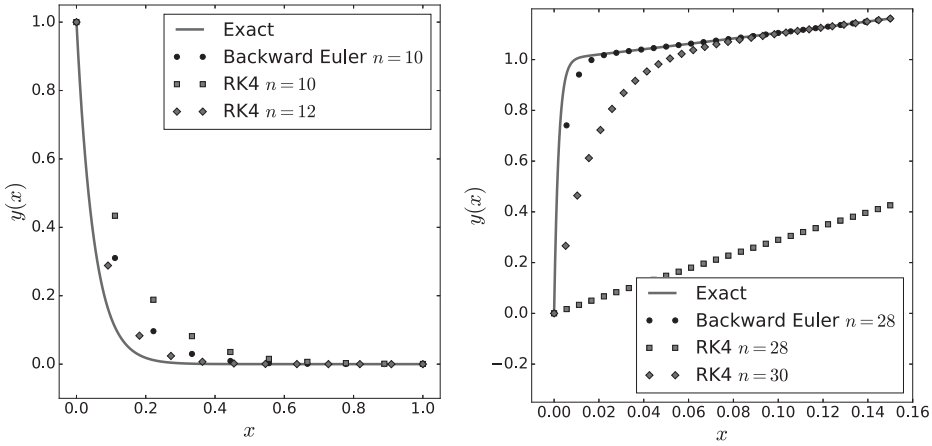
Stiffness

As mentioned earlier, the fourth-order Runge–Kutta method is fairly robust and should usually be your go-to solution. As another example of how well it does, we can solve the test equation for which, as we saw in our discussion of Fig. 8.2, the forward Euler method faced stability issues. The left panel of Fig. 8.5 shows the fourth-order Runge–Kutta and the backward Euler methods for $y'(x) = -20y(x)$. Overall, we see that RK4 is certainly decent: for $n = 10$ it does slightly more poorly than the backward Euler method, but qualitatively the behavior is the same for both $n = 10$ and $n = 12$. This doesn't really come as a surprise: RK4 has a local error of $O(h^5)$, so it can "get the most out" of a finite step size h; in other words, it does a dramatically better job than the forward Euler method and that's enough. Based on this panel, you might be tempted to think that our earlier digression on stability and implicit methods was pointless: you can use a high-order *explicit* method without having to worry about complicated root-finding steps.

Of course, this is too good to be true. In earlier chapters (especially in our discussion of linear algebra in chapter 4) we've seen that one must distinguish between the *stability* of a given method and the *ill-conditioning* of a given problem. There are well-conditioned problems which some methods fail to solve (we say they are *unstable*) and other methods manage to solve (we say they are *stable*). On the other hand, there exist ill-conditioned problems, which no method can hope to attack very fruitfully. In between these two extremes, there are mildly ill-conditioned problems which can be hard to solve even if you're employing a generally stable method. The ill-conditioning of a differential equation is known as *stiffness*: a *stiff* ODE makes many methods unstable. Quantifying stiffness in the general case can get quite complicated (but see below for more on this), so let's look at a specific example to see stiffness in action.

Here's an example that will put our earlier RK4-oriented triumphalism to rest:

$$y'(x) = 501e^x - 500y(x), \qquad y(0) = 0 \tag{8.72}$$



RK4 and backward Euler applied to the test equation (left) and our stiff problem (right) **Fig. 8.5**

As usual, this is a first-order initial-value problem; its exact solution is $y(x) = e^x - e^{-500x}$. The exact solution is already warning us of what could go wrong: one needs to keep track of two very different scales: a growing exponential that has x in the exponent and a decaying exponential that has $-500x$ in the exponent. Roughly speaking, stiff equations will always exhibit this feature of needing to keep track of two different scales at the same time. As you may have imagined, in order to be able to handle both scales, one needs to employ a step size h that is short enough to resolve the smallest of the scales in our problem: of course, that means that the calculation is likely to be pretty expensive.

This discussion is made concrete in the right panel of Fig. 8.5, where we show the backward Euler and RK4 methods applied to Eq. (8.72). We find that, even for a small number of points ($n = 28$) the backward Euler method does a reasonably good job. On the other hand, we find that RK4 behaves strangely: the global error of this Runge–Kutta method is supposed to be $O(h^4)$, but when we change $n = 28$ to $n = 30$ (i.e., go from $h = 0.0056$ to $h = 0.0052$) we, once again, encounter a dramatic change in behavior. In short, RK4 becomes unstable at some point, even though the (implicit) backward Euler method has no trouble handling the same problem. While it is true that if you "throw more points at the problem" RK4 will behave properly (i.e., converge), the figure shows that sometimes even this solid technique gets in trouble. Of course, Eq. (8.72) was hand-picked to emphasize the problems that arise when dealing with stiff ODEs. In the problem set you will encounter other, more innocuous-looking, cases; pretty often, such issues arise when you have to deal with two (or more) exponentials or two (or more) oscillatory behaviors. You can sometimes transform your troubles away, i.e., employ an analytical trick *before* you apply a technique such as RK4.

A general point should be made here: in these figures we have been employing a small number of points ($n = 10$ or $n = 30$). In practical applications, you typically use more points; this is fine if the evaluation of your right-hand side f is trivial but, as is sometimes the case, if your ODE itself is the result of another computation, it may be very costly to use, say, $n = 1000$ points or more. As always, one tries to employ as few points as

possible, given one's accuracy requirements. The point we've been making/repeating in this section is that if you find yourself dealing with a stiff ODE, then you also have to take into consideration stability requirements in addition to accuracy requirements. Of course, you might not know ahead of time that your ODE is stiff, so some care is needed when determining the number of points used in a numerical integrator.

Adaptive Stepping

Our discussion of stiff ODEs and our need to resolve the finer scale involved in our problem raises the wider question of how many points we should use in our numerical integration. In earlier sections we've (implicitly) made two assumptions: (a) we can determine which n is good enough by manually checking a few cases, and (b) the same step size h will be employed throughout the integration region. As the reader may have already guessed, both of these assumptions are not tenable in heavy-duty applications: it would be nice to have an automatic routine, which determines on its own how many steps to use; it would be even better if such a routine can sub-divide the problem into "easy" and "hard" regions, distributing the total number of points appopriately. In this section, we remove each of these assumptions in turn, slowly ramping up the complications that have to be introduced into the algorithm. The problem set asks you to recover our results by implementing such adaptive integration schemes yourself.

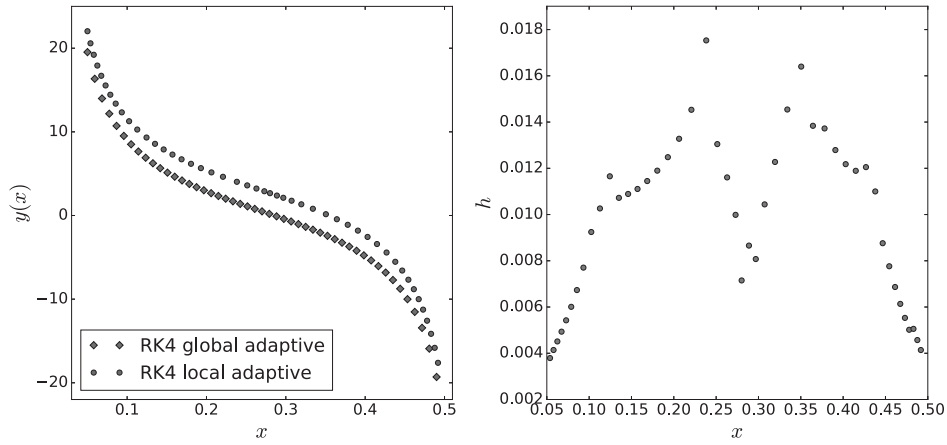
Global Adjustment

The simplest tack is completely analogous to our adaptive-integration schemes in section 7.3, where we were doubling the number of panels.[11] To refresh your memory, what we did there was to carry out a calculation employing N panels and another one for $N' = 2N$ panels. We then took advantage of the fact that the error in the composite Simpson's rule is ch_N^4 in the first case and $ch_{N'}^4$ in the second. After a brief derivation, we were able to show that the error in the calculation employing $N' = 2N$ panels can be expressed in terms of the difference between the two estimates (employing N and N' panels, respectively). Crucially, this was a derivation for the *composite* Simpson's rule: in the first case we were using $n = N - 1$ points and in the second case $n' = N' - 1$ points; as per Eq. (7.65), the two numbers are related by $n' = 2n - 1$. Both n and n' were expected to get quite large, depending on the error budget one was dealing with.

Repeating that derivation for the present case (of integrating an IVP from a to b) would entail first carrying out a calculation using n points (where the best approximation to $y(b)$ would be y_{n-1}) and then another calculation using n' points (where the best approximation to $y(b)$ would be $y_{n'-1}$); as before, $n' = 2n - 1$. Observe that y_{n-1} and $y_{n'-1}$ correspond to two distinct calculations. Since the global error for the RK4 method, just like the error in the composite Simpson's rule, is $O(h^4)$, the derivation is completely analogous:

$$y(b) = y_{n-1} + ch_N^4 = y_{n'-1} + ch_{N'}^4 \tag{8.73}$$

[11] Step halving is also used in ODE solvers that employ Richardson extrapolation, as you will discover when you solve the relevant problem.



Global and shifted local adaptive RK4 (left) and the step size for the local case (right) **Fig. 8.6**

This, together with $h_N = 2h_{N'}$, can be manipulated to eliminate $h_{N'}$, thereby giving:

$$\mathcal{E} = \frac{1}{15}\left(y_{n'-1} - y_{n-1}\right) \tag{8.74}$$

where the 15 in the denominator is a direct consequence of using RK4. Note that this is the absolute error; you can divide with $y_{n'-1}$ (i.e., the best of the two estimates) if you want to approximate the relative error. Just like for Simpson's rule, we have here been doing calculations with many points which are equally spaced from a to b.

As you will find out when you solve the relevant problem, applying this scheme to our nonlinear Riccati equation from Eq. (8.71) leads to the points shown with diamonds in the left panel of Fig. 8.6. The trend is identical to that in Fig. 8.4, only this time we are not manually choosing a specific value of n: instead, we have pre-set a specific *global* tolerance for our approximation to $y(b)$ and then keep doubling the number of panels until we meet our accuracy goals. The advantage of such an adaptive scheme is that we don't have to worry too much about the details of our problem: the algorithm will keep on using more points until the error tolerance goals are met.

Local Adjustment

As should be immediately obvious from the distribution of our diamond points, the global adaptive scheme can be quite wasteful: it employs equally spaced points and doubles the total number of panels used every time, without regard to the behavior of the solution. In other words, this scheme has no way of accounting for the possibility that our function may exhibit lots of structure in one region but have almost nothing going on elsewhere: the points are always equally spaced in x. This is highlighted by how our diamonds are placed for this example near a and b: since this is where our function shows most of its structure,

the points appear to be farther apart (in y) than they are elsewhere, where they appear tightly packed. This behavior ties in to our earlier comments on stiff ODEs: if you're using the same step size everywhere, and you wish to effectively capture two different length scales, you will be forced to use a tiny step size everywhere.[12]

Thus, with a view to being non-wasteful, we should come up with a way of *locally* adjusting the step size h. The formalism ends up looking quite similar to what we did in the previous subsection, but conceptually what we're up to is quite different: we will start at a given point x_j and take one or two steps away from it. Specifically, we can take a single step from x_j to $x_j + h$, getting the following relation for the exact solution $y(x_j + h)$:

$$y(x_j + h) = \tilde{y}_{j+1} + \kappa h^5 + O(h^6) \tag{8.75}$$

Let's take a moment to unpack the notation here: $\tilde{y}_{j+1}$ is the result of starting at the point x_j, y_j, and taking a step as per Eq. (8.64): we're using a tilde to emphasize that this is a single step, but other than that we haven't done anything new; you can think of y_j as $\tilde{y}_j$ if it helps; we use Eq. (8.64) to produce $\tilde{y}_{j+1}$ from $\tilde{y}_j$, implicitly assuming that we knew the starting value exactly. We're explicitly showing the local error for this one step, κh^5, and then capturing all higher-order terms with $O(h^6)$.

Now picture taking two steps instead of one: first, from x_j to $x_j + h/2$ and then from $x_j + h/2$ to $x_j + h$. The error for the first step will look like:

$$y\left(x_j + \frac{h}{2}\right) = \tilde{\tilde{y}}_{j+1/2} + \kappa\left(\frac{h}{2}\right)^5 + O(h^6) \tag{8.76}$$

where we are using double-tildes to emphasize that in this scenario we will be taking two steps to go from x_j to $x_j + h$. In Eq. (8.76) we used Eq. (8.64) to go from y_j to $\tilde{\tilde{y}}_{j+1/2}$. After we take the second step we will have:

$$y\left(x_j + h\right) = \tilde{\tilde{y}}_{j+1} + 2\kappa\left(\frac{h}{2}\right)^5 + O(h^6) \tag{8.77}$$

where now there is an extra factor of 2 in the error, because we didn't actually start from $y(x_j + h/2)$ but from $\tilde{\tilde{y}}_{j+1/2}$; we are assuming that κ stays constant from x_j to $x_j + h$, which is likely true for small steps.

If we now equate Eq. (8.75) and Eq. (8.77), we find:

$$\tilde{\tilde{y}}_{j+1} - \tilde{y}_{j+1} = \frac{15}{16}\kappa h^5 \tag{8.78}$$

which clearly shows that the difference between our two estimates, $\tilde{\tilde{y}}_{j+1} - \tilde{y}_{j+1}$, is proportional to h^5. We can now plug this relation back in to Eq. (8.77) to get:

$$y\left(x_j + h\right) = \tilde{\tilde{y}}_{j+1} + \frac{1}{15}\left(\tilde{\tilde{y}}_{j+1} - \tilde{y}_{j+1}\right) + O(h^6) \tag{8.79}$$

[12] The exact solution to Eq. (8.72) consists of two exponentials with little overlap, so it's not obvious why one should use the same step size everywhere.

This quantifies the error in our best estimate of the value $y\left(x_j + h\right)$. Observe that both Eq. (8.74) and Eq. (8.79) involve the number 15, though the origins are not quite the same: in order to derive the first we dealt with global errors which are $O(h^4)$, whereas for the second we used the local error which is $O(h^5)$. Furthermore, the two function values we are subtracting in Eq. (8.79) both correspond to the point $x_j + h$, whereas in Eq. (8.74) we were dealing with estimates of the function at $x_{n-1} = x_{n'-1} = b$.

We don't have to stop here, though: we can take advantage of the fact that the difference of the two estimates scales as h^5, in order to produce a guess for the *next* step size! In other words, for a given h we get a $\tilde{\tilde{y}}_{j+1} - \tilde{y}_{j+1}$ that is proportional to h^5. We can reverse the argument, to ask: which $\bar{h}$ should we use if we want the estimate-difference to be equal to an absolute tolerance, Δ? In equation form:

$$\frac{|\tilde{\tilde{y}}_{j+1} - \tilde{y}_{j+1}|}{\Delta} = \left(\frac{h}{\bar{h}}\right)^5 \tag{8.80}$$

This relationship can be solved for $\bar{h}$; we choose to be conservative and multiply the result of this equation with a *safety factor*, $\alpha < 1$:

$$\bar{h} = \alpha h \left|\frac{\Delta}{\tilde{\tilde{y}}_{j+1} - \tilde{y}_{j+1}}\right|^{0.2} \tag{8.81}$$

The safety factor ensures that we make the estimate even smaller, just in case; typical values used are 0.8 or 0.9. As a secondary precaution, one can choose to employ Eq. (8.81) only if it's telling us to increase the step size by a factor of, say, up to 5: this ensures that we don't have huge jumps in the magnitude of the step size.

Let's interpret what's going on here. In the right-hand side of Eq. (8.79) we've seen a way to quantify the error in $\tilde{\tilde{y}}_{j+1}$. Then, Eq. (8.81) tells us how to use the scaling of Eq. (8.78) in order to guess the next step size: (a) if $\tilde{\tilde{y}}_{j+1} - \tilde{y}_{j+1}$ is larger in magnitude than the desired tolerance Δ, then this equation tells us how to pick a new step size before re-trying the current step (from x_j to $x_j + h$); the step size will be *decreasing*, to allow us to have better chances at succeeding this time around, and (b) if $\tilde{\tilde{y}}_{j+1} - \tilde{y}_{j+1}$ is smaller in magnitude than the desired tolerance Δ, then that means that we've already met the tolerance requirement for the present step and can now turn to the next step (from $x_j + h$ to $x_j + 2h$); in that scenario, Eq. (8.81) tells us by how much we can *increase* the step size for that next step (except when α reduces it slightly). The only thing we've left out is which h to pick for the very first step: the short answer is that it doesn't really matter; if our first guess is too large, the algorithm will keep applying Eq. (8.81) until the tolerance test is successful.

One of the problems asks you to implement this local adaptive scheme using RK4 for the usual nonlinear Riccati equation. The results are shown in the left panel of Fig. 8.6: note that the two sets of results (global and local) lie on the same curve, but we have artificially shifted one set in order to help you distinguish it from the other. We have chosen the tolerances (global and local, respectively) such that both approaches give rise to roughly 50 points in total. The most prominent feature is that the circles are *not* equally spaced in

x: as advertised, they have been placed according to the behavior of the function we are solving for. As a consequence of this, we see that the large spacing (in y) that was exhibited by the global method near a and b is *not* a feature of the local method: our algorithm has figured out that the step size needs to be small in those regions and therefore placed more points there. You will also notice a larger density of points near the middle of the plot, but more diluteness on either side of the middle. Once again, this is the algorithm itself determining whether it needs a large or small step size, according to the solution's trend. To drive this point home, the right panel of Fig. 8.6 shows the step size h corresponding to each point in x:[13] you can immediately see that this is fully consistent with the statements we made in connection with the left panel. Crucially, for this example, we see a variation of more than a factor of 4: the local adaptive method dedicates fewer points and therefore spends less time in regions where nothing exciting is taking place.

Before concluding this section, we note that we have only discussed the simplest possible approaches to (global and local) adaptive integration; both of these employed RK4, which has been the best method we've seen so far. In the problem set you will encounter higher-order methods, including RK5; it is possible to use a combination of RK4 and RK5 as a way of quantifying the local error, instead of the single-step and double-step calculations employed above.

8.2.4 Simultaneous Differential Equations

We've spent quite a few pages up to this point discussing initial-value problems, all of which have had the form of Eq. (8.8), namely they have been first-order ordinary differential equations. The motivation behind this is that everything we've learned so far can also be applied to systems of simultaneous first-order differential equations (with many dependent variables) with minimal modification. As a matter of fact, even single ODEs of higher order can be cast as such systems; as a result, the techniques we've introduced (such as fourth-order Runge–Kutta) can be very straightforwardly generalized to solve much more complicated initial-value problems. Let us take some time to unpack these statements.

Two Equations

In physical problems, it is quite common that one has to solve two ordinary differential equations together: take the independent variable to be x (as before) and the two independent variables to be $y_0(x)$ and $y_1(x)$; note that there is nothing discretized going on here: $y_0(x)$ and $y_1(x)$ are functions of x which we must solve for. An example of such coupled ODEs is given by the TOV problem which we discussed at the start of this chapter. In the general case, the two ODEs we need to solve are:

[13] This is always the *successful* step size, i.e., the one that was able to meet our tolerance requirement.

$$\begin{aligned} y_0'(x) &= f_0\,(x, y_0(x), y_1(x)) \\ y_1'(x) &= f_1\,(x, y_0(x), y_1(x)) \\ y_0(a) &= c_0, \qquad y_1(a) = c_1 \end{aligned} \tag{8.82}$$

Note that since we have two independent variables, $y_0(x)$ and $y_1(x)$, we also have two equations, two right-hand sides, f_0 and f_1, and two known initial-values, $y_0(a)$ and $y_1(a)$.

Crucially, the prototypical problem of a (single) second-order IVP, given in Eq. (8.10):[14]

$$w'' = f(x, w, w'), \qquad w(a) = c, \qquad w'(a) = d \tag{8.83}$$

can be re-cast as a set of two simultaneous first-order ODEs. To see that, examine the following two definitions:

$$\begin{aligned} y_0(x) &= w(x) \\ y_1(x) &= w'(x) = y_0'(x) \end{aligned} \tag{8.84}$$

Our first new function is equal to our starting $w(x)$ and the second new function is equal to $w(x)$'s derivative. In the last step of the second relation we used the first relation. We can now combine these two equations together with Eq. (8.83) to get:

$$\begin{aligned} y_0'(x) &= y_1(x) \\ y_1'(x) &= f\,(x, y_0(x), y_1(x)) \\ y_0(a) &= c, \qquad y_1(a) = d \end{aligned} \tag{8.85}$$

The first equation is simply the second relation in Eq. (8.84), written with the derivative on the left-hand side. The second relation is Eq. (8.83) expressed in terms of $y_0(x)$ and $y_1(x)$. Similarly, we've translated the initial values into our new language. All in all, Eq. (8.85) is precisely of the form of Eq. (8.82): observe that f_0 is in this case particularly simple, whereas f_1 is the f contained in the second-order ODE. Thus, we have validated our earlier claim that second-order equations can be seen as simultaneous first-order ones.

General Case

Everything we discussed in the previous subsection can be straightforwardly generalized to the case of ν coupled equations (or that of a single ODE of ν-th order). Specifically, Eq. (8.82) takes the form:

$$\begin{aligned} y_i'(x) &= f_i\,(x, y_0(x), y_1(x), \ldots, y_{\nu-1}(x)) \\ y_i(a) &= c_i, \qquad i = 0, 1, \ldots, \nu - 1 \end{aligned} \tag{8.86}$$

This is just begging to be converted into vector notation:

[14] We are calling the dependent variable $w(x)$ for clarity, but we could just as well have called it $y(x)$.

$$\begin{aligned} \mathbf{y}'(x) &= \mathbf{f}\,(x, \mathbf{y}(x)) \\ \mathbf{y}(a) &= \mathbf{c} \end{aligned} \tag{8.87}$$

where $\mathbf{y}(x)$ bundles together ν functions, $\mathbf{f}$ stands for ν right-hand sides, and $\mathbf{c}$ is a vector of ν numbers (the initial values). Keep in mind that x, a, and b are *not* vectors but plain scalars. Since a single ODE of ν-th order is just a special case of Eq. (8.87), you should assume that whatever we say below applies to that case, also.

Armed with our vector notation, we are now in a position to start discretizing. Instead of re-deriving all the methods for IVPs in ODEs that we introduced in earlier sections, we will merely observe that they all carry over if you make the obviously necessary modifications. For example, the backward Euler method of Eq. (8.40) now takes the form:

$$\begin{aligned} \mathbf{y}_{j+1} &= \mathbf{y}_j + h\,\mathbf{f}(x_{j+1}, \mathbf{y}_{j+1}), \qquad j = 0, 1, \ldots, n-2 \\ \mathbf{y}_0 &= \mathbf{c} \end{aligned} \tag{8.88}$$

Take a moment to realize that we are no longer dealing with exact solutions (which are functions of x) but with discretizations (which are approximations that exist only at specific grid points). Also, note that $\mathbf{y}_j$ is the approximate value of *all* ν functions at x_j. If you need to spell things out, you will have to employ some extra notation here: for example, $\mathbf{y}_0$ groups together all the function values at the initial endpoint; that means that it contains the components $(y_0)_0$, $(y_1)_0$, and so on. Crucially, n controls how many grid points we use from a to b, whereas ν tells us how many simultaneous equations we need to solve.

Similarly, the forward Euler method of Eq. (8.20) now becomes:

$$\begin{aligned} \mathbf{y}_{j+1} &= \mathbf{y}_j + h\,\mathbf{f}(x_j, \mathbf{y}_j), \qquad j = 0, 1, \ldots, n-2 \\ \mathbf{y}_0 &= \mathbf{c} \end{aligned} \tag{8.89}$$

Our workhorse, the fourth-order Runge–Kutta prescription of Eq. (8.64), turns into:

$$\begin{aligned} \mathbf{k}_0 &= h\,\mathbf{f}(x_j, \mathbf{y}_j) \\ \mathbf{k}_1 &= h\,\mathbf{f}\left(x_j + \frac{h}{2}, \mathbf{y}_j + \frac{\mathbf{k}_0}{2}\right) \\ \mathbf{k}_2 &= h\,\mathbf{f}\left(x_j + \frac{h}{2}, \mathbf{y}_j + \frac{\mathbf{k}_1}{2}\right) \\ \mathbf{k}_3 &= h\,\mathbf{f}\left(x_j + h, \mathbf{y}_j + \mathbf{k}_2\right) \\ \mathbf{y}_{j+1} &= \mathbf{y}_j + \frac{1}{6}\left(\mathbf{k}_0 + 2\mathbf{k}_1 + 2\mathbf{k}_2 + \mathbf{k}_3\right) \end{aligned} \tag{8.90}$$

Marvel at how straightforward our generalization is. It should come as no surprise that NumPy's array functionality is perfectly matched to such vector manipulations, a topic we turn to next.

ivp_two.py Code 8.2

```
import numpy as np

def fs(x,yvals):
     y0, y1 = yvals
     f0 = y1
     f1 = - (30/(1-x**2))*y0 + ((2*x)/(1-x**2))*y1
     return np.array([f0, f1])

def rk4_gen(fs,a,b,n,yinits):
     h = (b-a)/(n-1)
     xs = a + np.arange(n)*h
     ys = np.zeros((n, yinits.size))

     yvals = np.copy(yinits)
     for j,x in enumerate(xs):
          ys[j,:] = yvals
          k0 = h*fs(x, yvals)
          k1 = h*fs(x+h/2, yvals+k0/2)
          k2 = h*fs(x+h/2, yvals+k1/2)
          k3 = h*fs(x+h, yvals+k2)
          yvals += (k0 + 2*k1 + 2*k2 + k3)/6
     return xs, ys

if __name__ == '__main__':
     a, b, n = 0.05, 0.49, 12
     yinits = np.array([0.0926587109375, 1.80962109375])
     xs, ys = rk4_gen(fs,a,b,n,yinits)
     print(ys)
```

Implementation

We start this section with a confession: our earlier implementation example (the Riccati equation of Eq. (8.71)) was chosen because it can be re-purposed to exemplify many distinct questions. We already saw how to solve this nonlinear first-order initial-value problem in Code 8.1. We now see that that equation can be recast if one makes the following *Riccati transformation*: $y(x) = w'(x)/w(x)$, where $w(x)$ is a new function. Using this transformation, Eq. (8.71) turns into:

$$w''(x) = -\frac{30}{1-x^2}w(x) + \frac{2x}{1-x^2}w'(x)$$
$$w(0.05) = 0.0926587109375, \qquad w'(0.05) = 1.80962109375 \tag{8.91}$$

This is now a *second*-order ODE. Crucially the $y^2(x)$ has dropped out of the problem, meaning that this is now a *linear* ODE! Since it's of second order, we need to supply two initial values: $w(a)$ and $w'(a)$. It's time for a second confession: this is not just any old second-order linear ODE. It's the *Legendre differential equation*, namely the differential equation whose solution gives us Legendre polynomials; given the presence of the number 30 on the right-hand side (which is the result of 6×5), we expect our solution to be $P_5(x)$: this is how we have produced the initial values at $a = 0.05$, namely by using Code 2.8, i.e., `legendre.py`.[15]

We can now rewrite our problem in Eq. (8.91) in the form of two coupled first-order ODEs, as per Eq. (8.85):

$$y_0'(x) = y_1(x)$$
$$y_1'(x) = -\frac{30}{1-x^2}y_0(x) + \frac{2x}{1-x^2}y_1(x) \tag{8.92}$$
$$y_0(0.05) = 0.0926587109375, \qquad y_1(0.05) = 1.80962109375$$

This is now in sufficiently general form to allow us to solve it via minimal modifications of our earlier code.

Code 8.2 shows a Python implementation that solves this IVP problem of two simultaneous ODEs. Specifically, the function `fs` expects as input the x_j and a one-dimensional numpy array of two values, representing $\mathbf{y}_j$ at a given x_j. It evaluates the two right-hand sides in Eq. (8.92) and then returns its own array of two values. You may be thinking that this is too formal for a simple problem involving only two ODEs. The reason we've done things this way is because the other function in this program, `rk4_gen()`, is completely general: it is an implementation of Eq. (8.90) that works for the ν-dimensional problem just as well as for our two-dimensional problem here; as mentioned after that equation, NumPy functionality is very convenient, in that the code closely follows the vector relations. The specialization to the case of two equations is carried out only in the input parameter `yinits`, which bundles together the initial values, i.e., $\mathbf{y}_0$. This is completely analogous to what we did in Code 5.5, i.e., `multi_newton.py`, when implementing Newton's multidimensional root-finding method.[16] We employ an $n \times 2$ array to store all the $\mathbf{y}_j$'s together; we move to a new row each time we step on to the next x_j.

The main program sets up the integration interval and the number of points, as well as the initial values, calls our numerical integrator, and prints out the 12×2 `ys` array, suppressed here for brevity; if you're solely interested in the w_j's, you can look at the 0th column only. We're not plotting the result here because you already know what it looks like: it's

[15] Almost every chapter in this book has had some relation to Legendre polynomials!

[16] Note also that our implementation is a linear combination of the two previous subsections: `rk4_gen()` applies to the case of ν equations, but `fs()` to the case of two equations.

simply $P_5(x)$ from Fig. 2.6; as a matter of fact, the 1st column can also be benchmarked by comparing to $P_5'(x)$ in the right panel of that figure.

Stability and Stiffness for Simultaneous Equations

Having introduced and implemented the more general problem of ν simultaneous first-order ODEs, we now take some time to investigate the aforementioned concepts of stability and stiffness as they apply to this problem. You may recall that stability was introduced in the context of Euler's method and stiffness in the context of RK4, but these are much broader concepts: stability has to do with the method one is employing (Euler, RK4, etc.) while stiffness has to do with properties of the problem, which then impacts which methods are stable (and which aren't). We will now examine these two concepts in turn for the general problem of ν simultaneous ODEs.

First, we wish to discuss the idea of *stability* of a given method for the problem of ν simultaneous equations. We introduced stability (of a given method) by studying the test equation, Eq. (8.33). The most obvious way of generalizing this to ν equations is:

$$\mathbf{y}'(x) = \mathbf{A}\mathbf{y}(x), \qquad \mathbf{y}(0) = \mathbf{c} \tag{8.93}$$

where, for us, $\mathbf{A}$ is a $\nu \times \nu$ matrix made up of real numbers. This is the minimal modification to the test equation: it contains a first-order derivative on the left-hand side and the function itself (i.e., the functions themselves) on the right-hand side. Obviously, these are *coupled* ODEs: the first equation contains $y_0'(x)$ on the left-hand side but, due to the presence of $\mathbf{A}$ on the right-hand side, involves all the $y_0(x), y_1(x), \ldots, y_{\nu-1}(x)$. Note that this is a *linear* system of ODEs.

We would have preferred it if we had been faced with the problem where the ODE for $y_0'(x)$ only involves $y_0(x)$, the one for $y_1'(x)$ only $y_1(x)$, and so on. In short, we would like to *diagonalize* this problem. We immediately remember Eq. (4.173), which taught us how to diagonalize a matrix:[17] $\mathbf{A} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^{-1}$; here $\mathbf{\Lambda}$ is the diagonal "eigenvalue matrix" made up of the eigenvalues λ_i and $\mathbf{V}$ is the "eigenvector matrix", whose columns are the eigenvectors $\mathbf{v}_i$ (have a look at section 4.4 if you need a refresher). If we express $\mathbf{A}$ in this way and then multiply with $\mathbf{V}^{-1}$ on the left, our equation becomes:

$$\mathbf{V}^{-1}\mathbf{y}'(x) = \mathbf{\Lambda}\mathbf{V}^{-1}\mathbf{y}(x), \qquad \mathbf{V}^{-1}\mathbf{y}(0) = \mathbf{V}^{-1}\mathbf{c} \tag{8.94}$$

If we now introduce a new (vector) variable:[18]

$$\mathbf{z}(x) = \mathbf{V}^{-1}\mathbf{y}(x) \tag{8.95}$$

our equation takes the form:

$$\mathbf{z}'(x) = \mathbf{\Lambda}\mathbf{z}(x), \qquad \mathbf{z}(0) = \mathbf{V}^{-1}\mathbf{c} \tag{8.96}$$

[17] Incidentally, the present stability analysis requires the ability to handle nonsymmetric matrices.
[18] This entire argument is very similar to the derivation around Eq. (4.212).

or, in index notation:

$$z_i'(x) = \lambda_i z_i(x), \qquad z_i(0) = \left(\mathbf{V}^{-1}\mathbf{c}\right)_i, \qquad i = 0, 1, \ldots, \nu - 1 \tag{8.97}$$

We have accomplished what we set out to: $z_i'(x) = \lambda_i z_i(x)$ involves only a single function, so it is of the form of the original test equation, Eq. (8.33). This means that each of these equations can be immediately solved to give the analytical solution $z_i(x) = e^{\lambda_i x} z_i(0)$. We will only deal with real eigenvalues; thus, we see that if all the eigenvalues are *negative*, then all the $z_i(x)$'s will be decaying functions of x (i.e., $z_i(x)$ goes to 0 as x goes to infinity). From Eq. (8.95) we see that $\mathbf{y}(x) = \mathbf{V}\mathbf{z}(x)$, meaning that all the $y_i(x)$'s will also decay away.

Note that up to this point in this subsection we haven't actually studied any specific method, i.e., we were dealing with a system of ODEs, not with a discretization scheme. Let us now investigate a specific technique, namely the forward Euler method of Eq. (8.89); for the system in Eq. (8.93) this gives:

$$\mathbf{y}_{j+1} = \mathbf{y}_j + h\,\mathbf{A}\mathbf{y}_j \tag{8.98}$$

If we now repeatedly apply this equation from $j = 0$ to $j = n - 2$, we get:

$$\mathbf{y}_{n-1} = (\boldsymbol{\mathcal{I}} + h\mathbf{A})^{n-1}\mathbf{y}_0 \tag{8.99}$$

In order to get a decaying solution as $n \to \infty$, we need to impose as a condition that the eigenvalues of the matrix $\boldsymbol{\mathcal{I}} + h\mathbf{A}$ are all strictly less than one in magnitude. We learned this fact in our discussion of the power method, section 4.4.1; at the time this was an unwanted side-effect, but here we *want* the solution to decay. If λ_i are the eigenvalues of $\mathbf{A}$, then the eigenvalues of $\boldsymbol{\mathcal{I}} + h\mathbf{A}$ are simply $1 + h\lambda_i$; take a moment to see that this is true.

In other words, we have reached the requirement:

$$|1 + h\lambda_i| < 1 \tag{8.100}$$

for any i. If we now assume, as per Eq. (4.176), that our eigenvalues are sorted, with λ_0 being the dominant eigenvalue, we can formulate the following *stability condition*:

$$h < \frac{2}{|\lambda_0|} \tag{8.101}$$

If this condition is not satisfied, the forward Euler method will lead to a numerically unstable solution. It's worth pausing to appreciate that the eigenvalues of the matrix $\mathbf{A}$ appeared both in our study of the system of ODEs, Eq. (8.94), and in our analysis of the forward Euler method's stability, Eq. (8.101); in the former case, we saw that a decaying solution corresponds to all the eigenvalues being negative, whereas in the latter case it implied a specific constraint on the step size: it needs to be smaller than $2/|\lambda_0|$.

It is now time to discuss a *stiff* system of ODEs. Our earlier analysis tells us that we will need a step size that is very small if the largest eigenvalue is very large. You can see that if the eigenvalues are several orders of magnitude apart, we will need to step through our system with a tiny step determined by the largest eigenvalue, even though the exact solution might not be severely impacted by that term; otherwise, we risk our method

becoming unstable, at which point we don't trust its results at all. Let's look at a specific example:

$$\mathbf{y}'(x) = \begin{pmatrix} 98 & 198 \\ -99 & -199 \end{pmatrix} \mathbf{y}(x), \qquad \mathbf{y}(0) = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \tag{8.102}$$

The exact solution is:

$$y_0(x) = 2e^{-x} - e^{-100x}, \qquad y_1(x) = -e^{-x} + e^{-100x} \tag{8.103}$$

as you can verify by substitution. We immediately see that the exact solution contains two contributions of very different scales: e^{-100x} and e^{-x}. This is consistent with the fact that the two eigenvalues here are -100 and -1. The stability condition in Eq. (8.101) tells us that we need to take $h < 2/100$ if we want the forward Euler method to be stable: this, despite the fact that e^{-100x} is considerably smaller than e^{-x}, so we wouldn't have expected the presence of e^{-100x} to matter for reasonable values of x. This is a clear-cut case of a stiff system of ODEs: the two eigenvalues are orders of magnitude apart, thereby imposing a stringent stability step size, much beyond what we would have needed for accuracy purposes. As in earlier examples, implicit methods do a better job for such stiff problems; you will see this in action in one of the problems.

8.3 Boundary-Value Problems

As mentioned at the start of section 8.2, this chapter is not limited to initial-value problems; even so, the techniques we introduced earlier will also be helpful in what follows. Given this, the present section, tackling boundary-value problems, as well as the following one, on eigenvalue problems, will be shorter; the emphasis will be on implementation (of already known methods, this time for the new problems) and on introducing new implicit methods.

Our prototypical boundary-value problem will be a second-order ODE:

$$w'' = f(x, w, w'), \qquad w(a) = c, \qquad w(b) = d \tag{8.104}$$

where $w(a) = c$ and $w(b) = d$ are the boundary conditions; observe that we do *not* know $w'(a)$, so this is not an initial-value problem. In other words, we can't just start from $x = a$ and then step through the ODE using Euler, RK4, or any of the earlier methods. The f in Eq. (8.104) could be either linear or nonlinear; the latter case is harder, as you will discover in the problem set. Similarly, the boundary conditions specified could also be more complicated: what we have in Eq. (8.104), where the function's value is provided at the endpoint, $w(b)$, is known as a *Dirichlet boundary condition*. The problem set examines the alternative case, where the function-derivative's value is given at the endpoint, $w'(b)$, a scenario known as a *Neumann boundary condition*.

Below, we will discuss two general approaches to solving this BVP: (a) treat it as a

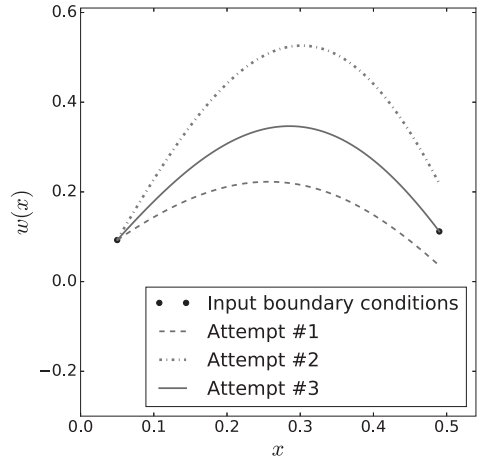


Fig. 8.7 Illustration of the shooting method for a boundary-value problem

more complicated version of an initial-value problem, or (b) face head-on the fact that we know two boundary conditions and set up a method accordingly. We discuss these two approaches in turn.

8.3.1 Shooting Method

We said (correctly) that Eq. (8.104) is not an initial-value problem, because we do not have all the necessary information to start integrating out from a. In other words, we can't blindly apply the material of section 8.2.4 to this problem. The simplest way to approach this problem is: what if we *could* treat this as an initial-value problem? It's true that we don't know $w'(a)$, but why not simply guess it and see if that works? This is illustrated in Fig. 8.7: what we *do* have as input are the values of $w(a)$ and $w(b)$, shown with circles in the figure. We then guess $w'(a)$ and see what happens. Obviously, a random guess is extremely unlikely to be able to satisfy the boundary condition $w(b)$; that's OK, though, because we can try again. In the example of this figure, it's easy to see that the value for $w'(a)$ we first chose was too small; we can simply adjust that to be larger and try again. The second attempt doesn't satisfy the boundary condition $w(b)$ either; we can keep playing this game, though. This time, we reduce the value of $w'(a)$ and try again. Eventually, this process will lead to a value of $w'(a)$ that is "just right", allowing us to go through the point $w(b)$, as desired. (The figure shows the, highly optimistic, scenario where it only took us three tries to get there.) This iterative procedure is obviously analogous to aiming at a target and, as a result, is known as the *shooting method*.

Let us try to recast the above prescription in a more systematic way. Our input is $w(a) = c$ and $w(b) = d$. In the language of our discretization scheme(s), these input conditions can be expressed as $w_0 = c$ and $w_{n-1} = d$. What we do not know is $w'(a)$, but let's assume that its (unknown) value is σ, i.e, $w'(a) = \sigma$. A moment's thought will convince you that you can take c and σ, plug them into a numerical integrator, like those in section 8.2.4,

and determine the corresponding w_{n-1} value that comes out. In other words, a method like RK4 for this problem transforms the input guess σ into a specific output at the right endpoint, w_{n-1}; mathematically, we can say that our integration method is a function that takes us from σ to w_{n-1}, i.e., $g(\sigma) = w_{n-1}$, where g is a new function describing the effect of solving our ODE with a given σ, from a to b. In this language, since our goal is that $w_{n-1} = d$, we can demand that:

$$g(\sigma) = d \tag{8.105}$$

To reiterate, g is not an analytically known function (like f, which is the right-hand side in Eq. (8.104)); instead, it is the result of carrying out a full initial-value problem solution from a to b. Then, what we are faced with in Eq. (8.105) is a *root-finding* problem, like those we spent so much time studying in chapter 5. The function g may be linear or nonlinear, but this is irrelevant, since we know already how to solve a general algebraic equation; as a matter of fact, f itself may be linear or nonlinear, but this doesn't really change how the shooting method works. To summarize, the shooting method combines initial-value solvers like forward Euler with iterative solvers like Newton's method: for each guess of σ an IVP needs to be solved. This confirms our earlier claim that boundary-value problems are more complicated to solve than initial-value problems.

Implementation

A code implementing the shooting method will fuse two elements: a numerical integrator and a root-finder. As already mentioned, this is a general feature, regardless of whether or not the ODE we are faced with is linear. For the sake of concreteness, we will here tackle the Legendre differential equation (again) that, in its latest incarnation, was used to illustrate the solution of two simultaneous differential equations, Eq. (8.91). This time around, instead of providing $w(a)$ and $w'(a)$ like before, our input will be $w(a)$ and $w(b)$:

$$\begin{gathered} w''(x) = -\frac{30}{1-x^2}w(x) + \frac{2x}{1-x^2}w'(x) \\ w(0.05) = 0.0926587109375, \qquad w(0.49) = 0.1117705085875 \end{gathered} \tag{8.106}$$

If you're wondering where we got the value of $w(0.49)$, the answer is that we already know the solution: we know this should be the Legendre polynomial $P_5(x)$, so we can simply use `legendre.py`. Alternatively, we can look at the output of running `ivp_two.py`. Put another way, we already know what $w'(a)$ should be in order to get the $w(0.49)$ shown in Eq. (8.106): we could simply look at the input provided in `ivp_two.py`. Of course, these avenues would not be open to us in general: we would simply be given the ODE and the boundary values and would have to solve for $w'(a)$. In other words, in what follows we ignore all prior knowledge other than what is shown in Eq. (8.106).

Code 8.3 bvp_shoot.py

```
from secant import secant
from ivp_two import fs, rk4_gen
import numpy as np

def shoot(sig):
     a, b, n = 0.05, 0.49, 100
     yinits = np.array([0.0926587109375, sig])
     xs, ys = rk4_gen(fs,a,b,n,yinits)
     wfinal = 0.11177050858750004
     return ys[-1, 0] - wfinal

if __name__ == '__main__':
     wder = secant(shoot,0.,1.)
     print(wder)
```

Code 8.3 shows a Python implementation of the shooting method for our problem. From a practical perspective, we plan to re-use two of our earlier codes: we are faced with a second-order differential equation, so we'll use `rk4_gen()` from `ivp_two.py`; as a matter of fact, since the ODE hasn't changed, we can also re-use `fs()` from the same code. The only thing that will be different this time around is the input: we (pretend to) no longer know both of the elements of `yinits` that `rk4_gen()` expects to receive as input. We will also need a general root-finding function; Newton's method requires information on the derivative, so it's simply easier to reach for the secant method from `secant.py`. Other than importing these three functions, our code doesn't actually do very much: we define a new function, `shoot()`, which plays the role of $g(\sigma) - d$ in Eq. (8.105). It does this by carrying out an RK4 solution for the given σ argument and then subtracting the desired $w(b)$ from the value of w_{n-1} it has produced. Make sure you understand why we wrote `ys[-1, 0]`: the first index corresponds to the discretization of the x_j's and the second index to the equation we are solving, see Eq. (8.85).

The main program simply calls `secant()`, giving it `shoot()` as the function whose root we're after. As you may recall, the secant method requires two initial guesses to get going, so we arbitrarily provide it with two numbers; as you will discover when you run this code, these don't really matter: in any case, this code solves $g(\sigma) = d$ very quickly, giving us the $w'(a)$ that respects our desired boundary condition $w(b)$.

8.3.2 Matrix Approach

Let's do a brief recap: the shooting method approaches the second-order ODE boundary-value problem of Eq. (8.104) by casting it as a set of two simultaneous ODEs, guessing one initial value, and then combining an IVP solver (for simultaneous first-order ODEs)

with a root-finder (for algebraic equations in one variable). We now turn to an alternative approach, which tackles our second-order ODE directly: to see what this means, remember that we started our discussion of IVP methods in section 8.2 by discussing the forward and backward Euler methods; these were motivated by finite-difference approximations to the *first* derivative, since that's what was involved in the ODE we were faced with at the time. Here, we will follow a similar route: use a finite-difference approximation for both the *second* and the *first* derivative involved in Eq. (8.104).

To be explicit, we first repeat our problem from Eq. (8.104):

$$w'' = f(x, w, w'), \qquad w(a) = c, \qquad w(b) = d \tag{8.107}$$

We now reach for the central-difference approximation to the first and second derivatives, for points on a grid. That is precisely what we studied in section 3.3.6, so all we need to do is to apply Eq. (3.38) and Eq. (3.39), thereby transforming our problem into:

$$\begin{aligned} \frac{w_{j-1} - 2w_j + w_{j+1}}{h^2} &= f\left(x_j, w_j, \frac{w_{j+1} - w_{j-1}}{2h}\right), \qquad j = 1, 2, \ldots, n-2 \\ w_0 &= c, \qquad w_{n-1} = d \end{aligned} \tag{8.108}$$

Note that this set of equations involves only the w_j's, i.e., there are no derivatives left. Posing and solving these equations is a process that is completely different from what we were doing for IVPs above: we *cannot* start at the left endpoint and work our way to the right one; instead, we need to solve for *all* of these w_j values at the same time. Another way of putting the same fact: w_{j-1}, w_j, and w_{j+1} appear on both the left-hand side and the right-hand side, bringing to mind the implicit methods we encountered earlier (only this time it's not just y_{j+1} on both sides). These features will become clearer in the following subsection, where we discuss a specific example.

The approach given in Eq. (8.108) is known by many names in the literature, e.g., *finite-difference method* or *equilibrium method*. In order to avoid any confusion, and so as to emphasize its difference from the shooting method, we here keep things general and call it the *matrix approach* to BVPs. Observe that we haven't said much about f so far: if this is linear (as in our example below), then Eq. (8.108) gives rise to a linear system of equations (with a *single* solution), of the type that we spent much of chapter 4 solving (section 4.3); if, on the other hand, f is nonlinear, then we would be faced with a system of nonlinear equations which, as we learned in section 5.4, is a formidable task, especially if you don't have a decent initial guess for the (entire) solution, as is usually the case. Since we need to solve for all the w_j's simultaneously, you can see that taking, say, $n = 100$ for the nonlinear case is already a highly non-trivial task. It might help to recall that the shooting method employed a root-finder regardless of the linearity of f; however, that was a root-finding problem in a single variable and therefore reasonably straightforward. If you're wondering why you would need to use such a large n with the matrix approach, you may benefit from recalling the truncation error involved in our finite-difference approximations for the first and second derivatives, Eq. (3.38) and Eq. (3.39); in both cases, the error was $O(h^2)$, which is considerably worse than the $O(h^5)$ exhibited by RK4. On the other hand, the

matrix approach satisfies the boundary condition *exactly*, which reflects its better *stability* properties (barring the extreme case where the matrix involved is singular).

Matrix Method Applied to Our Equation

This might all be a tad too abstract for you (e.g., what *matrix* are we referring to?); to shed some light, let us apply Eq. (8.108) to a specific example. To keep things simple, we will study the BVP of the (linear) Legendre differential equation, Eq. (8.106), again. For this case, the right-hand side of Eq. (8.107) takes the form:

$$f(x, w, w') = -\frac{30}{1-x^2}w(x) + \frac{2x}{1-x^2}w'(x) \tag{8.109}$$

We can now plug this equation into Eq. (8.108) to find:

$$\begin{gathered} w_{j-1} - 2w_j + w_{j+1} = -h^2\frac{30}{1-x_j^2}w_j + \frac{hx_j}{1-x_j^2}(w_{j+1} - w_{j-1}), \qquad j = 1, 2, \ldots, n-2 \\ w_0 = 0.0926587109375, \qquad w_{n-1} = 0.1117705085875 \end{gathered} \tag{8.110}$$

The equation on the first line involves w_{j-1}, w_j, and w_{j+1} on both the left-hand side and on the right-hand side. If we group terms appropriately, we can re-express this problem as:

$$\begin{gathered} w_0 = 0.0926587109375 \\ \alpha_j w_{j-1} + \beta_j w_j + \gamma_j w_{j+1} = 0, \qquad j = 1, 2, \ldots, n-2 \\ w_{n-1} = 0.1117705085875 \end{gathered} \tag{8.111}$$

where we've defined:

$$\alpha_j = 1 + \frac{hx_j}{1-x_j^2}, \quad \beta_j = -2 + \frac{30h^2}{1-x_j^2}, \quad \gamma_j = 1 - \frac{hx_j}{1-x_j^2} \tag{8.112}$$

in order to make the notation less cumbersome. Now, Eq. (8.111) can be straightforwardly re-expressed in matrix format:

$$\begin{pmatrix} 1 & 0 & 0 & \ldots & 0 & 0 & 0 \\ \alpha_1 & \beta_1 & \gamma_1 & \ldots & 0 & 0 & 0 \\ 0 & \alpha_2 & \beta_2 & \ldots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & \beta_{n-3} & \gamma_{n-3} & 0 \\ 0 & 0 & 0 & \ldots & \alpha_{n-2} & \beta_{n-2} & \gamma_{n-2} \\ 0 & 0 & 0 & \ldots & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_{n-3} \\ w_{n-2} \\ w_{n-1} \end{pmatrix} = \begin{pmatrix} 0.0926587109375 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \\ 0.1117705085875 \end{pmatrix} \tag{8.113}$$

where you can, finally, see why this is called the *matrix* approach. Crucially, Eq. (8.113) is of the form $\mathbf{Ax} = \mathbf{b}$: the n quantities w_j are unknown and everything else in this equation is already known. As noted above, you can't simply step through the elements from left to right (or from right to left). You might benefit from writing out the $n = 6$ case explicitly: w_0 is immediately known; the next row gives an equation relating w_1 to w_2, neither of which

bvp_matrix.py Code 8.4

```
from gauelim_pivot import gauelim_pivot
import numpy as np

def matsetup(a,b,n):
     h = (b-a)/(n-1)
     xs = a + np.arange(n)*h

     A = np.zeros((n,n))
     np.fill_diagonal(A, -2 + 30*h**2/(1-xs**2))
     A[0,0] = 1; A[-1,-1] = 1
     np.fill_diagonal(A[1:,:], 1 + h*xs[1:]/(1-xs[1:]**2))
     A[-1,-2] = 0
     np.fill_diagonal(A[:,1:], 1 - h*xs/(1-xs**2))
     A[0,1] = 0

     bs = np.zeros(n)
     bs[0] = 0.0926587109375
     bs[-1] = 0.11177050858750004
     return A, bs

def riccati(a,b,n):
     A, bs = matsetup(a, b, n)
     ws = gauelim_pivot(A, bs)
     return ws

if __name__ == '__main__':
     a, b, n = 0.05, 0.49, 400
     ws = riccati(a, b, n); print(ws)
```

is known; the row after that an equation involving w_1, w_2, and w_3, but we still have more unknowns than equations. However, if you write out *all* the corresponding equations then you *can* solve them together; after all, this is a linear system of equations. Here we merely wanted to emphasize that the matrix approach to the BVP is conceptually different from how we used finite differences in section 8.2 to produce a given function value from the previous one (remember: we were studying *single-step* techniques).

Our matrix equation, Eq. (8.113), can be solved via, say, Gaussian elimination. Of course, this is a *tridiagonal* coefficient matrix, so applying a general-purpose method like Gaussian elimination is quite wasteful; this is why we introduced the Jacobi itera-

tive method, the Gauss–Seidel method, or an LU method which was tailored to tridiagonal matrices in the main text and the problem set of chapter 4. Here we are not interested in the details of the implementation or its efficiency, but in the big-picture concepts differentiating the shooting method from the matrix approach to BVPs.

Implementation

Code 8.4 is a Python implementation of Eq. (8.113). Most of the code is inside a single function, `matsetup()`, which sets up the coefficient matrix and the constant vector. Just like in earlier codes, this is accomplished by slicing the arrays we pass as the first argument to `numpy.fill_diagonal()`. We manually set some special values as needed. The only other function in this code, `riccati()`, calls `matsetup()` and then our earlier Gaussian-elimination-with-pivoting function from chapter 4. This is all quite straightforward. The main program sets up the problem and then calls `riccati()`. The (standard) notation here is potentially confusing: a and b are the (scalar) endpoints, while $\mathbf{A}$ and $\mathbf{b}$ (a matrix and a vector, respectively) define the system of Eq. (8.113). In other words, `a` is not `A` and, similarly, `b` is not `bs`.

Keep in mind that this time around `ws` is a 1d array (as opposed to `ys` in `bvp_shoot.py`, which was a 2d array). This is as it should be, since Eq. (8.113) was set up by discretizing Eq. (8.107) directly, i.e., without having to introduce a new function corresponding to the derivative values. We don't display the output, since it closely matches what we encountered earlier: this is the same Legendre differential equation solution we've been studying for the last few sections. Of course, a few minor tweaks in how you set up the matrix would allow you to study totally different problems on your own.

8.4 Eigenvalue Problems

We now turn to a specific subclass of boundary-value problems, which is sufficiently distinct that it gets its own section. This is the case where the boundary-value problem's equation involves (in addition to x, $w(x)$, $w'(x)$, and $w''(x)$) a parameter s: this one parameter, known as the *eigenvalue*, will make a world of difference in how this problem is tackled. Our prototypical eigenvalue problem will be a second-order ODE:

$$w'' = f(x, w, w'; s), \qquad w(a) = c, \qquad w(b) = d \tag{8.114}$$

This looks an awful lot like Eq. (8.104); the only difference is the presence of s. This means that we are not faced with a *single* differential equation, but a *family* of ODEs. For given values of s there may be no solution to the corresponding ODE; thus, we will have to compute the "interesting" values of s (i.e., the ones that lead to a solution) in addition to producing an approximation to $w(x)$. The f contained in Eq. (8.114) could be quite complicated, but then the EVP typically becomes intractable. We will here focus on the special case of a linear second-order differential equation such as:

$$w''(x) = \zeta(x)w'(x) + \eta(x)w(x) + \theta(x)\ s\ w(x) \tag{8.115}$$

where the functions $\zeta(x)$, $\eta(x)$, and $\theta(x)$ are known. If we also impose homogeneous boundary conditions, this is (close enough to) a *Sturm–Liouville form*.

We could now proceed to see how to solve this general problem. The obvious candidates are the techniques we introduced in the previous section on BVPs: the shooting method and the matrix approach. Especially in the latter case, the details of the derivation become cumbersome; we opt, instead, to study a specific example of a linear second-order eigenvalue problem, for which we will show all the details explicitly. The problem set guides you to study other physical scenarios, including some drawn from quantum mechanics. Our problem of choice will be the *Mathieu equation*, which appears in the study of string vibrations and a host of other topics in physics:

$$w''(x) = (2q\cos 2x - s)\, w(x), \qquad w(0) = w(2\pi) \tag{8.116}$$

We didn't actually specify the values of the function at the two boundaries; we merely stated that the solution is periodic. You should not be surprised that the solution (can) be periodic: the equation contains a $\cos 2x$ term, so it is certainly plausible that that "drives" a corresponding periodicity in $w(x)$; we will see below that the solution to this ODE does not *have* to be periodic, in which case it is not a solution to our eigenvalue problem (since it doesn't obey the boundary condition we have imposed).

The s parameter will be very important below: we will need to find the values of s that lead to non-trivial (i.e., non-zero) solutions for $w(x)$. The q parameter determines the strength of the cosine term and will be held fixed (at a finite value) in what follows. Take a moment to appreciate that when $q = 0$ this ODE takes the form $w''(x) + sw(x) = 0$, on which you have spent a good chunk of your undergraduate education. The mere fact that we are now exploring possible values of q and s shows that this problem is qualitatively different from what we were studying earlier in this chapter. Let us see how to solve the Mathieu equation in practice.

8.4.1 Shooting Method

As you may recall, the shooting method amounts to guessing a value of $w'(a)$, solving an IVP problem, and seeing if you managed to hit upon $w(b)$. Obviously, this process can be systematized, as we saw earlier when we employed the secant root-finding method to produce a better guess for $w'(a)$, by finding the root(s) of Eq. (8.105), $g(\sigma) = d$. Unfortunately, this approach doesn't trivially generalize to the problem at hand. To see that, let's look at the Mathieu equation again:

$$w''(x) = (2q\cos 2x - s)\, w(x), \qquad w(0) = w(2\pi) = 0 \tag{8.117}$$

This time around we took $w(0) = w(2\pi) = 0$, i.e., we are searching for odd solutions.[19]

[19] As we'll find out soon enough, the solutions will have a period that is *at most* 2π.

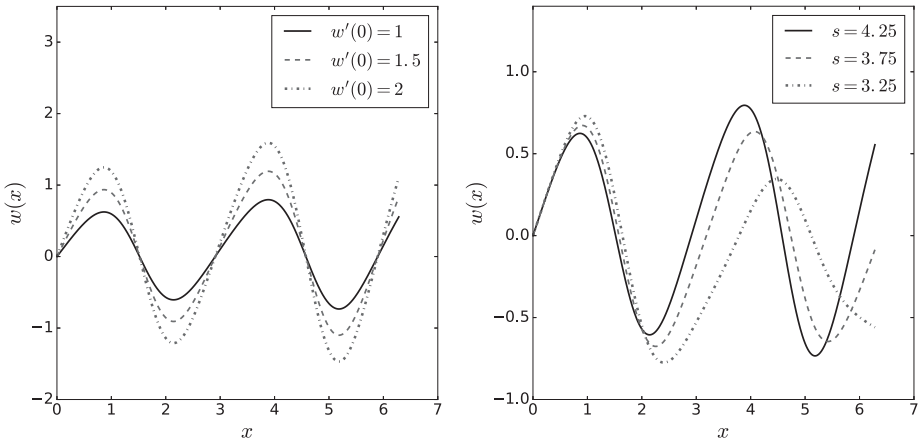


Fig. 8.8 Mathieu solutions for $q = 1.5$: $s = 4.25$ (left) and $w'(0) = 1$ (right)

One problem with applying the shooting method as described above to this equation is that we don't actually know the value of s. But let's assume you do. Even then, you will have trouble using the solution of the IVP to get the derivative at the first endpoint. Let's see why. For this linear ODE, if $w(x)$ is a solution then so is $\alpha w(x)$, where α is an arbitrary constant. Observe that if $w(0) = w(2\pi)$ then $\alpha w(0) = \alpha w(2\pi)$ also holds. But that means that trying out different values of $w'(0)$ won't get us anywhere, since the first derivative of $\alpha w(x)$ at $x = 0$ is $\alpha w'(0)$ and α could be anything. As you may know from a course on quantum mechanics, determining the α in $\alpha w(x)$ is a process known as *normalization*, which has nothing to do with actually satisfying our boundary conditions (since those are satisfied regardless of the value of α). If you haven't studied quantum mechanics yet, there's no need to worry: we are merely stating that you will need an external requirement if you wish to find α.

If you think this is too vague, look at the left panel of Fig. 8.8: we've arbitrarily picked q and s. We see that, for our choice of s, the boundary condition is simply not met (here and below we keep $q = 1.5$ fixed). Then, by changing the value of $w'(0)$ all we're accomplishing is to multiply the entire solution by a constant: if it didn't meet the boundary condition for one initial value $w'(0)$, then no amount of stretching is going to save us. In other words, the value of $w'(0)$ is related to the normalization, but doesn't help us accomplish our goal. The right panel of the same figure shows that it is s which might help us actually satisfy the boundary conditions in Eq. (8.117): this time around we know that the value of $w'(0)$ is irrelevant, so we keep it fixed. We see that as we tune s we get close to satisfying $w(2\pi) = 0$; obviously, we need to come up with a systematic way of doing so.

Our goal can be accomplished by slightly tweaking our earlier argument: instead of using a root-finder to determine $w'(0)$ (a hopeless task), arbitrarily pick $w'(0)$ and then use a root-finder to determine s. Our input boundary conditions $w(0) = w(2\pi) = 0$ can, for our discretization scheme(s), be expressed as $w_0 = w_{n-1} = 0$. As explained, we are free to randomly guess a value of $w'(0) = \sigma$. What we do next is to take our starting w_0 and σ, plug them into a numerical integrator, like those in section 8.2.4, and determine the

corresponding w_{n-1} value that comes out. For a randomly picked value of s this approach is fated to fail: for a given s, a method like RK4 transforms the input s into a specific output at the right endpoint, w_{n-1}, without caring about the "desired" value of $w_{n-1} = 0$; mathematically, we can say that our integration method is a function that takes us from s to w_{n-1}, i.e., $g(s) = w_{n-1}$, where g is a new function describing the effect of solving our ODE with a given s, with x going from 0 to 2π. This g is (analogous to, but) different from that appearing in Eq. (8.105): the latest g is a function of the s parameter in our Mathieu equation. In this language, since our goal is that $w_{n-1} = 0$, we can demand that:

$$g(s) = 0 \tag{8.118}$$

As before, g is the result of carrying out a full initial-value problem solution from 0 to 2π. Once again, we can apply a root-finding algorithm to solve this equation; the only difference is that this time we are not searching for a starting derivative value, but for the value of s for which our boundary condition $w(2\pi) = 0$ can be met.

Implementation

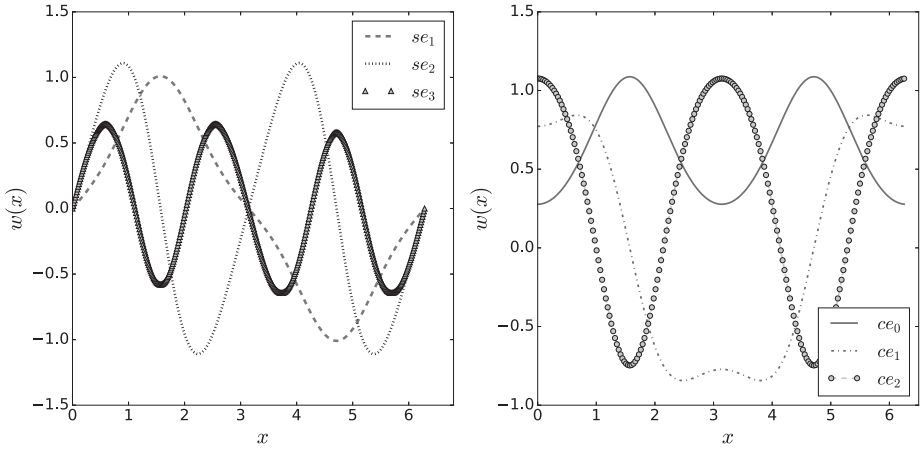
Implementing the shooting method for eigenvalue problems should be simple enough. After all, we're still combining the same tools as in the BVP case: a numerical integrator for an IVP (like RK4) and a root-finder (like the secant method). Just as in Fig. 8.8, we are setting the cosine strength to $q = 1.5$, but it's totally straightforward to explore other values later. Thinking of how to proceed, our first thought is that Code 8.3, i.e., `bvp_shoot.py` might be a good template. However, things are not so simple. In that program we imported `fs()` and `rk4_gen()` from `ivp_two.py`, which made sense at the time, since we were faced with a well-defined second-order ODE, namely the Legendre differential equation of Eq. (8.106). In the present case, this won't work: even keeping q fixed, we cannot avoid the fact that our differential equation itself depends on the value of the s parameter. As a result, `rk4_gen()` cannot be used, since it assumes the entire differential equation is given (i.e., not parameter dependent).

One could consider using a closure here, "binding" one parameter to fixed values for selected purposes. Instead, in Code 8.5 we follow the quick-and-dirty route, writing new versions of the f and RK4 routines. This still follows the general philosophy of casting a second-order ODE as two first-order ODEs, as per Eq. (8.85), but this time our f and RK4 functions take in an extra parameter corresponding to s. This allows us, in the new version of `shoot()`, to solve for s by determining the value that satisfies Eq. (8.118), i.e., the one that leads to $w_{n-1} = 0$. As advertised, we pass in an arbitrary value of $w'(0)$ which shouldn't (and doesn't) affect anything other than the normalization.

In the main program, we provide a few initial guesses and then print out the corresponding solutions (for the s's). The output of running this code is:

Code 8.5 evp_shoot.py

```
from secant import secant
import numpy as np

def fs(x,yvals,s):
     q = 1.5
     y0, y1 = yvals
     f0 = y1
     f1 = (2*q*np.cos(2*x) - s)*y0
     return np.array([f0, f1])

def rk4_gen_eig(fs,a,b,n,yinits,s):
     h = (b-a)/(n-1)
     xs = a + np.arange(n)*h
     ys = np.zeros((n, yinits.size))

     yvals = np.copy(yinits)
     for j,x in enumerate(xs):
          ys[j,:] = yvals
          k0 = h*fs(x, yvals,s)
          k1 = h*fs(x+h/2, yvals+k0/2,s)
          k2 = h*fs(x+h/2, yvals+k1/2,s)
          k3 = h*fs(x+h, yvals+k2,s)
          yvals += (k0 + 2*k1 + 2*k2 + k3)/6
     return xs, ys

def shoot(s):
     a, b, n = 0, 2*np.pi, 500
     yinits = np.array([0., 5.])

     xs, ys = rk4_gen_eig(fs,a,b,n,yinits,s)
     wfinal = 0.
     return ys[-1, 0] - wfinal

if __name__ == '__main__':
     for sinit in (-0.4, 3.3, 8.5):
          sval = secant(shoot,sinit,sinit+0.5)
          print(sval, end=" ")
```



Eigenfunctions for the Mathieu problem: odd (left) and even (right) **Fig. 8.9**

```
-0.73326514905    3.81429091649    9.09260876706
```

where we have suppressed the intermediate print-outs from `secant()`. We had implied this before, but now have irrefutable evidence: there is more than one value of s for which we can solve our eigenvalue problem (i.e., there is more than one eigenvalue).

Each eigenvalue (i.e., each value of s for which we can satisfy the boundary conditions non-trivially) corresponds to a solution for $w(x)$; unsurprisingly, these are known as *eigenfunctions*. In the left panel of Fig. 8.9 we are plotting the three eigenfunctions corresponding to the three eigenvalues of the output. We've taken the opportunity to label these eigenfunctions according to their standard names from the theory of Mathieu functions, but this doesn't really matter for our purposes. What *does* matter is that these are periodic functions that look *almost* like sines, but not quite; all three of them are odd functions of x. The difference from a sine is more pronounced in the case of se_1. Note that one of these curves has period of π and the other two a period of 2π; of course, even se_2, which has a period of π, still satisfies the desired boundary condition, $w(0) = w(2\pi) = 0$. This happens to be the solution that the right panel of Fig. 8.8 was moving toward. You should play around with the main program, trying to find the next eigenvalue/eigenfunction pairs.

Intriguingly, we have computed these eigenvalues and eigenfunctions by building on our earlier concepts: no complicated theory needed to be invoked. Even better: our techniques are of much wider applicability than the one problem of the Mathieu equation.

8.4.2 Matrix Approach

We just saw how to tweak the idea of the shooting method developed for the BVP, in order to have it work for the EVP. In the present subsection we'll do something analogous for the matrix approach. In other words, similarly to what we did in Eq. (8.108), we can directly tackle the second-order ODE of Eq. (8.114), replacing the second and first derivatives with the corresponding central-difference approximations. Once again, this is most easily

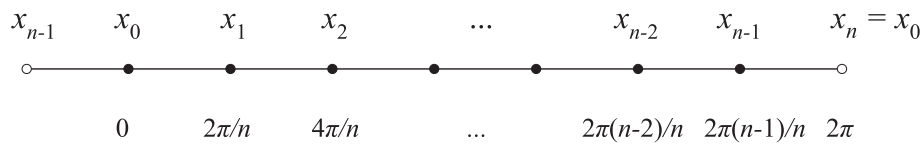


Fig. 8.10 Grid for the x_j's, with solid dots showing our actual points and open dots implied ones

grasped for a specific case; thus, we will now apply such a discretization scheme to the Mathieu equation of Eq. (8.116), repeated here for your convenience:

$$w''(x) = (2q\cos 2x - s)\, w(x), \qquad w(0) = w(2\pi) \tag{8.119}$$

This time around, we will *not* assume that $w(0) = w(2\pi) = 0$ as we did for the shooting method. Rather, we will only enforce that the solution be periodic, thereby getting both the odd functions seen above, as well as some new (even) functions.

We wish to build the periodicity in to our method. To do so, we recall that we had faced a similar situation in chapter 6: specifically, in Eq. (6.83) we set up a grid accordingly:

$$x_j = \frac{2\pi j}{n}, \qquad j = 0, 1, \ldots, n-1 \tag{8.120}$$

Observe that, unlike the grid of Eq. (8.15), here we include the left endpoint but not the right endpoint: this is because we know our function is periodic, so we avoid storing needless information. Of course, as always, we are still dealing with n points; this is illustrated in Fig. 8.10. Take a moment to compare to what we did for the shooting method: there we started at the left endpoint, 0, and explicitly integrated all the way up to the right one, 2π, in order to check if our solution has the desired value there. In the present case, we will always satisfy the boundary condition *exactly*. Of course, just as in the case of the BVP, this pleasant feature has to be weighed against the unfortunate detail that the truncation error involved in our finite-difference approximations for the first and second derivatives, Eq. (3.38) and Eq. (3.39), is $O(h^2)$, which is much worse than the $O(h^5)$ exhibited by RK4.

The Mathieu equation involves $w''(x)$ and $w(x)$, but not $w'(x)$: this means we only need to use the central-difference approximation to the *second* derivative, Eq. (3.39). Doing so leads to the following set of equations:

$$\frac{w_{j-1} - 2w_j + w_{j+1}}{h^2} = \Big(2q\cos 2x_j - s\Big)\, w_j, \qquad j = 0, 1, \ldots, n-1 \tag{8.121}$$

We can now group terms, to get:

$$w_{j-1} + \alpha_j w_j + w_{j+1} = -h^2 s w_j, \qquad j = 0, 1, \ldots, n-1 \tag{8.122}$$

where we defined:

$$\alpha_j = -2 - 2h^2 q\cos 2x_j \tag{8.123}$$

for ease of reading. Observe that both in Eq. (8.121) and in Eq. (8.122), we made no mention of the boundary terms. If you actually write down Eq. (8.122) for $j = 0$ and

$j = n - 1$, you are faced with the pesky terms w_{-1} and w_n which don't actually exist, if you take our grid in Eq. (8.120) seriously. We interpret these in the most natural way possible:

$$w_{-1} \equiv w_{n-1}, \qquad w_n \equiv w_0 \tag{8.124}$$

These definitions are fully consistent with the spirit of Fig. 8.10. With that clarification in mind, we are now in a position to write out the n equations of Eq. (8.122) in matrix form:

$$\begin{pmatrix} \alpha_0 & 1 & 0 & \dots & 0 & 0 & 1 \\ 1 & \alpha_1 & 1 & \dots & 0 & 0 & 0 \\ 0 & 1 & \alpha_2 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & \alpha_{n-3} & 1 & 0 \\ 0 & 0 & 0 & \dots & 1 & \alpha_{n-2} & 1 \\ 1 & 0 & 0 & \dots & 0 & 1 & \alpha_{n-1} \end{pmatrix} \begin{pmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_{n-3} \\ w_{n-2} \\ w_{n-1} \end{pmatrix} = -h^2 s \begin{pmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_{n-3} \\ w_{n-2} \\ w_{n-1} \end{pmatrix} \tag{8.125}$$

Unmistakably, this is of the form $\mathbf{Av} = \lambda\mathbf{v}$, eliminating any lingering doubts you may have had about why this is called an *eigenvalue problem*. Seeing the role that s plays in this equation similarly justifies why the s in Eq. (8.116) is called an *eigenvalue*.

Take a moment to compare this equation with our result for the matrix approach as applied to the BVP, Eq. (8.113), which was of the form $\mathbf{Ax} = \mathbf{b}$. There, the right-hand side was known, whereas here the unknown w_j's appear on both the left-hand side and the right-hand side. The $n \times n$ matrix on the left-hand side of Eq. (8.125) is cyclic tridiagonal, similar to the one we ecountered in the problem set of chapter 4. You should think about the origin of those two units (at the bottom left and top right). Gratifyingly, our final result allows us to apply the eigenvalue/eigenvector methods we spent much of chapter 4 (section 4.4) developing. As advertised, we are *building in* the periodicity, but not making any assumptions about the value of the solution at the endpoints (i.e., we are not limited to homogeneous boundary conditions). Finally, Eq. (8.125) allows us to produce the *spectrum* of eigenvalues, not just the lowest one (or few); using n points you produce approximations for n eigenvalues. This deserves to be emphasized: in the shooting method you need to zero-in on candidate eigenvalues manually; you may end up getting the same solution more than once, or you might accidentally skip over a given solution (a fact you would try to determine by counting nodes/zeros in the eigenfunctions). In contradistinction to this, the matrix approach gives you estimates for n eigenvalues. In a problem, you will explore if it does equally well for all eigenvalues.

Implementation

Code 8.6 is an implementation of the matrix approach to the eigenvalue problem for the Mathieu equation. Just like in Code 8.4, most lines are dedicated to a function setting up the coefficient matrix, `matsetup()`. This implements Eq. (8.125), once again using `numpy.fill_diagonal()` and a couple of manual assignments. A separate function, `mathieu()`, calls `matsetup()` and then `qrmet()` from chapter 4 to apply the QR method

Code 8.6 evp_matrix.py

```
from qrmet import qrmet
import numpy as np

def matsetup(q,n):
     h = 2*np.pi/n
     xs = np.arange(n)*h

     A = np.zeros((n,n))
     np.fill_diagonal(A, -2 - 2*h**2*q*np.cos(2*xs))
     np.fill_diagonal(A[1:,:], 1)
     np.fill_diagonal(A[:,1:], 1)
     A[0,-1] = 1
     A[-1,0] = 1
     return A

def mathieu(q,n):
     A = matsetup(q, n)
     qreigvals = qrmet(A,200)
     h = 2*np.pi/n
     qreigvals = np.sort(-qreigvals/h**2)
     return qreigvals

if __name__ == '__main__':
     q, n = 1.5, 200
     qreigvals = mathieu(q, n)
     print(qreigvals[:6])
```

for eigenvalue evaluation. We then divide with $-h^2$ and sort so that we can extract the s's in order. As expected, the quality of the output (for a given finite-difference approximation) gets better as the number of points n increases; of course, given the (lack of) efficiency in our implementation of the QR method, this slows things down, practically speaking. Even so, it's typically worth the wait since, as noted, our output consists of the first n eigenvalues.

The main program sets up the cosine strength, q, and the number of points on our grid, n, and calls `mathieu()`. We limit the output to the first six eigenvalues:

```
[-0.93706036 -0.73350696 2.16553604 3.81267332 4.74538385 9.08581402]
```

Every other number here is (close to) the output of Code 8.5, i.e., `evp_shoot.py`: we

have produced *all* the eigenvalues; half correspond to odd eigenfunctions and half to even eigenfunctions. We have plotted the latter in the right panel of Fig. 8.9, where you can see that two have a period of π and one has a period of 2π; these look somewhat like cosines but, especially in the case of ce_1, they are clearly different. Note that this approach can be used to produce the left panel of Fig. 8.9, as well. While we didn't explicitly say so, the Mathieu equation also arises in quantum mechanics (the "quantum pendulum").

8.5 Project: Poisson's Equation in Two Dimensions

Having spent the entire chapter up to this point studying *ordinary* differential equations, we now turn to *partial* differential equations; this is a huge subject, deserving of its own book. To accurately solve PDEs, you typically require specialized techniques, tailored to a given equation. In the spirit of the rest of this volume, in this chapter we've been examining general techniques of wide applicability; several of these can be (and are) also applied to PDEs. Instead of ending the volume on that uninspiring note, however, we here first provide some general comments about different types of PDEs, then choose a specific one, and solve it using techniques introduced in chapter 6. This charming example is meant to emphasize the significance of discrete Fourier transforms in all of scientific computing.

8.5.1 Examples of PDEs

Formally speaking, partial differential equations are usually divided into three classes: *hyperbolic*, *parabolic*, and *elliptic*. In order to keep things simple, we now briefly discuss a few specific examples of partial differential equations that are very important in physics:

- A typical example of a *hyperbolic* PDE is the *wave equation* involving $\phi(x,t)$:

$$\frac{\partial^2\phi}{\partial t^2} - c^2\frac{\partial^2\phi}{\partial x^2} = 0 \tag{8.126}$$

  where $c > 0$ and we assumed one spatial dimension. This equation involves a second-order time derivative and a second-order space derivative, which have opposite signs.
- A typical example of a *parabolic* PDE is the *heat equation* involving $\phi(x,t)$:

$$\frac{\partial\phi}{\partial t} - \alpha\frac{\partial^2\phi}{\partial x^2} = 0 \tag{8.127}$$

  where $\alpha > 0$ and we assumed one spatial dimension. This equation involves a first-order time derivative and a second-order space derivative, with opposite signs.[20]
- A typical example of an *elliptic* PDE is *Poisson's equation* involving $\phi(x,y)$:

$$\frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} = f(x,y) \tag{8.128}$$

[20] The time-dependent *Schrödinger equation*, Eq. (8.5), almost fits the bill; however, (a) it contains a pesky i on the left-hand side instead of having opposite signs, and (b) the wave function $\Psi(x,t)$ is, in general, *complex*. It's more appropriate to introduce a fourth class, *dispersive* PDEs, to capture these features.

where we assumed two spatial dimensions. Note that this involves two second-order spatial derivatives, which come in with the same sign.

Here's another way of looking at the classification of PDEs: Eq. (8.126) and Eq. (8.127) describe time evolution, which starts at a given point; thus, such dynamical equations correspond to what we called *initial-value problems*. On the other hand, Eq. (8.128) does not involve time; such a static situation involves specified boundary conditions, thereby giving rise to a *boundary-value problem*. In the interest of being concrete, we will now focus on the latter scenario, namely the boundary-value problem involving Poisson's equation.

8.5.2 Poisson's Equation via FFT

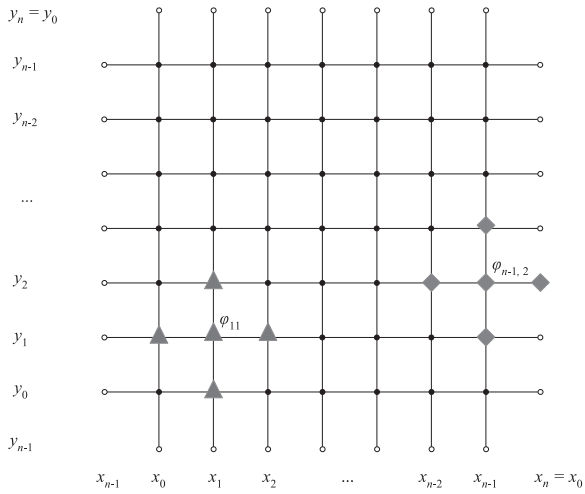
In a course on electromagnetism, you've seen Poisson's equation, Eq. (8.128), with the right-hand side $-\rho(x,y)/\epsilon_0$ where ϵ_0 is the permittivity of free space, $\rho(x,y)$ is a specified electric charge density, and our dependent variable is the electric potential $\phi(x,y)$; we're studying the two-dimensional case in order to make the visualizations easier, but there's no *a priori* reason keeping you from tackling the three-dimensional case. Thus, this problem is very closely related to the projects of chapters 1 and 2 at the start of the book, where we saw how to visualize fields and carry out the multipole expansion, respectively; the problem set discusses such "textbook" approaches to Poisson's equation. Finally, you may know that, in addition to appearing in electromagnetism, Poisson's equation is also very significant to astrophysics, in which case ϕ is the gravitational potential and the right-hand side takes the form $-4\pi G\rho(x,y)$ (again in two spatial dimensions), with $\rho(x,y)$ being the mass density and G Newton's gravitational constant. Regardless of the physical application, mathematically speaking our problem is a constant-coefficient elliptic equation and we'll also specialize it further, studying only *periodic* boundary conditions.

Problem and Discretization

In order to be concrete, we will tackle the following specific problem:

$$\begin{gathered}\frac{\partial^2\phi}{\partial x^2}+\frac{\partial^2\phi}{\partial y^2}=f(x,y), \qquad f(x,y)=\cos(3x+4y)-\cos(5x-2y) \\ \phi(x,0)=\phi(x,2\pi), \qquad \phi(0,y)=\phi(2\pi,y)\end{gathered} \tag{8.129}$$

As advertised, this is a periodic problem, both in its boundary conditions and in its forcing term. You may be thinking that you can solve this problem analytically, in which case you may not see the point of using a computer; but what if you were dealing with Mathieu functions instead of cosines? The computational tools we'll introduce below will apply even if the right-hand side of Poisson's equation is not an analytically known function, as you will discover when you tackle the problem set.



Two-dimensional periodic grid, with examples **Fig. 8.11**

We are now ready to discretize, placing our points on a grid:[21]

$$x_p = \frac{2\pi p}{n}, \qquad p = 0, 1, \ldots, n-1 \tag{8.130}$$

where we are building in the fact that our problem is periodic: the last point is at $2\pi(n-1)/n$, not at 2π, because we already know that all quantities have the same value at 2π as at 0. Since we're dealing with a two-dimensional problem, we also need to discretize the y's and therefore employ another index:

$$y_q = \frac{2\pi q}{n}, \qquad q = 0, 1, \ldots, n-1 \tag{8.131}$$

We have chosen to use the same discretization scheme for both coordinates: they both go from 0 to 2π, involving n points for each dimension. This is simply because that's what we'll need in what follows, but you will go beyond this in the problem set.

Similarly to what we did for the BVP and EVP, we now apply the central-difference approximation to the second-derivative (once for the x's and once for the y's) of Eq. (3.39):

$$\frac{\phi_{p+1,q} - 2\phi_{p,q} + \phi_{p-1,q}}{h^2} + \frac{\phi_{p,q+1} - 2\phi_{p,q} + \phi_{p,q-1}}{h^2} = f_{pq}, \qquad p, q = 0, 1 \ldots, n-1 \tag{8.132}$$

which can be slightly manipulated into the form:

$$\phi_{p+1,q} + \phi_{p-1,q} + \phi_{p,q+1} + \phi_{p,q-1} - 4\phi_{pq} = h^2 f_{pq} \tag{8.133}$$

Again, p and q are spatial indices, as per Eq. (8.130) and Eq. (8.131). Our two-dimensional grid is illustrated in Fig. 8.11, which generalizes Fig. 8.10 for our case; as before, solid circles show actual points and open circles are implied points. Each point ϕ_{pq} enters the

[21] Since n is already the number of points and i is the imaginary unit, we have to introduce new symbols.

discretized Poisson's equation, Eq. (8.133), together with its four nearest neighbors.[22] The figure also shows a couple of examples, highlighting the fact that our setup works equally well if ϕ_{pq} is somewhere in the middle of the grid, or right at its edge. This is not always the case: very often in the study of PDEs the implementation of the boundary conditions has to be handled separately (and can be a headache). Here, as in several other places throughout this volume, we study a periodic problem (because these show up in physics all the time) and therefore tailor our solution to the problem at hand.

You can solve Eq. (8.133) with linear algebra machinery, just like we did earlier: it is nothing other than a finite-difference result for a PDE. However, notice that the problem here is more complicated than before: for example, in Eq. (8.113) we were faced with an equation of the form $\mathbf{Ax} = \mathbf{b}$; here our unknowns are the ϕ_{pq}'s that together make up a *matrix*, not a vector! Of course, there are ways of organizing our n^2 equations in such a way that we can solve one problem of the form $\mathbf{Ax} = \mathbf{b}$ at a time; this is what you would encounter in a course on the numerical solution of PDEs. At its core, this is not a new method, just a messier version of what we saw earlier. Instead, we here opt for a totally different approach which solves Eq. (8.133) using the fast Fourier transform; in addition to being a nice application of the material we introduced in chapter 6, this also happens to be a very efficient approach, which obviates solving linear systems of equations altogether.

Solution via DFT

For the sake of (re)orientation, we remind you that in Eq. (8.133) we know the values of the f_{pq}'s on the right-hand side: these are simply the result of evaluating the $f(x, y)$ in Eq. (8.129) at the x_p's and y_q's given by Eq. (8.130) and Eq. (8.131). What we are trying to determine are the ϕ_{pq}'s on the left-hand side. Keep in mind that the left-hand side in Eq. (8.133) is basically the Laplacian operator in discretized coordinates. As you may recall from Fourier analysis (see chapter 6), derivatives turn into multiplications in Fourier (wave number) space. That means that we can be hopeful the complicated left-hand side in Eq. (8.133) will be dramatically simplified if we turn to the (two-dimensional) discrete Fourier transform; we haven't seen this before, but it is a straightforward generalization of the one-dimensional case. At a big-picture level, we are about to trade the p, q, which are spatial indices, with k, l, which are going to be Fourier indices. As mentioned, we are willing to go to Fourier space in order to simplify the effect of the derivative(s).

Generalizing Eq. (6.99) to the two-dimensional case allows us to write the inverse discrete Fourier transform of the discretized ϕ as follows:

$$\phi_{pq} = \frac{1}{n^2} \sum_{k=0}^{n-1} \sum_{l=0}^{n-1} \tilde{\phi}_{kl} e^{2\pi i k p/n} e^{2\pi i l q/n}, \qquad p, q = 0, 1 \ldots, n-1 \tag{8.134}$$

where we assumed we are using the same number of points, n, for each of the two dimensions in the problem, as above. We can do the same thing for the function appearing on the

[22] This is sometimes called a *five-point stencil*, though we generally prefer the term "discretization scheme".

right-hand side of Poisson's equation, namely the discretized f:

$$f_{pq} = \frac{1}{n^2} \sum_{k=0}^{n-1} \sum_{l=0}^{n-1} \tilde{f}_{kl} e^{2\pi i k p/n} e^{2\pi i l q/n}, \qquad p, q = 0, 1, \ldots, n-1 \tag{8.135}$$

Plugging these two relations into the equation we wish to solve, Eq. (8.133), we get:

$$\frac{1}{n^2} \sum_{k=0}^{n-1} \sum_{l=0}^{n-1} e^{2\pi i k p/n} e^{2\pi i l q/n} \left[\tilde{\phi}_{kl} \left(e^{2\pi i k/n} + e^{-2\pi i k/n} + e^{2\pi i l/n} + e^{-2\pi i l/n} - 4 \right) - h^2 \tilde{f}_{kl} \right] = 0 \tag{8.136}$$

At this point, we can use the orthogonality of complex exponentials in the discrete case, Eq. (6.92), to eliminate the sums. Explicitly, we multiply with $e^{-2\pi i k' p/n}$ and then sum over p and, similarly, multiply with $e^{-2\pi i l' q/n}$ and then sum over q. If we also replace the four exponentials inside the parentheses with two cosines, we are led to:

$$\tilde{\phi}_{kl} = \frac{1}{2} \frac{h^2 \tilde{f}_{kl}}{\cos(2\pi k/n) + \cos(2\pi l/n) - 2} \tag{8.137}$$

Thus, we can produce the $\tilde{\phi}_{kl}$ by taking the $\tilde{f}_{kl}$ and plugging them into Eq. (8.137).

We're almost done: since we've already computed $\tilde{\phi}_{kl}$, all that's left is for us to use the inverse DFT in Eq. (8.134) to evaluate the real-space ϕ_{pq}. That was our goal all along and we reached it without having to explicitly solve a linear system of n^2 coupled equations as per Eq. (8.133). The one thing we skipped over is that Eq. (8.137) involves $\tilde{f}_{kl}$ on the right-hand side, not the f_{pq} which is actually our input. That's OK, though, because we can simply use the following definition of the *direct* DFT in the two-dimensional case, which is a generalization of Eq. (6.97):

$$\tilde{f}_{kl} = \sum_{p=0}^{n-1} \sum_{q=0}^{n-1} f_{pq} e^{-2\pi i k p/n} e^{-2\pi i l q/n}, \qquad k, l = 0, 1, \ldots, n-1 \tag{8.138}$$

Our strategy is: (a) take f_{pq}, (b) use it to get $\tilde{f}_{kl}$ from the direct DFT in Eq. (8.138), (c) use that to evaluate $\tilde{\phi}_{kl}$ from Eq. (8.137), and (d) take the inverse DFT in Eq. (8.134) to get ϕ_{pq}.

Implementation

Thinking about how to implement the above approach to solving Poisson's equation, we realize that we don't actually have an implementation of a two-dimensional FFT (or even DFT, for that matter) available.[23] This is not a major obstacle: we will simply use the one-dimensional FFT implementation of Code 6.4, i.e., `fft.py`, to build up a two-dimensional version. In order to see how we could do that, we return to Eq. (8.138) and group the terms

[23] Of course, `numpy` has one but, as usual, it's more educational to write our own. Incidentally, in this section we use DFT and FFT interchangeably: the latter is the efficient implementation of the former, as always.

in a more suggestive manner:

$$\tilde{f}_{kl} = \sum_{p=0}^{n-1} \left[\sum_{q=0}^{n-1} f_{pq} e^{-2\pi i l q/n} \right] e^{-2\pi i k p/n} \tag{8.139}$$

As the brackets emphasize, it is possible to first take the one-dimensional DFT of each row (i.e., f_{pq} with p held fixed each time), and then take the one-dimensional DFT of each column of the result. Having applied the one-dimensional DFT algorithm $2n$ times, we have accomplished what we set out to do.

The only thing left is to figure out how to implement the two-dimensional *inverse* DFT when we have an implementation of the two-dimensional direct DFT at our disposal (as per the previous paragraph). We saw how to code up the one-dimensional inverse DFT in a problem in chapter 6; the idea generalizes straightforwardly to the two-dimensional case. Take the complex conjugate of the inverse DFT in Eq. (8.135):

$$f^*_{pq} = \frac{1}{n^2} \sum_{k=0}^{n-1} \sum_{l=0}^{n-1} \tilde{f}^*_{kl} e^{-2\pi i k p/n} e^{-2\pi i l q/n}, \qquad p, q = 0, 1, \ldots, n-1 \tag{8.140}$$

Thus, the direct DFT of $\tilde{f}^*_{kl}$, divided by n^2, is simply f^*_{pq}. But that means that, in order to produce the inverse DFT, we can simply: (a) take the complex conjugate of $\tilde{f}_{kl}$, (b) carry out the direct DFT of that, (c) divide by n^2, and (d) take the complex conjugate at the end.

Code 8.7 starts by importing the one-dimensional FFT functionality of `fft.py`. It then uses the procedure described above to define two functions that carry out the direct and inverse two-dimensional FFT. Note how both of these functions are very easy to write, given the Python and NumPy syntax and functionality. Another function is defined to provide the $f(x, y)$ from the right-hand side of Poisson's equation.

The physics is contained in the longest function in this program, called `poisson()`. After defining a one-dimensional grid of points, this function calls `numpy.meshgrid()` to produce two coordinate matrices: these are created in such a way that taking the `[p,q]` element from `Xs` and the `[p,q]` element from `Ys` gives you the pair (x_p, y_q).[24] The rows of `Xs` repeat the x_p's and the columns of `Ys` repeat the y_q's. In short, `numpy.meshgrid()` has allowed us to produce coordinate matrices, which we pass into `func()` to produce the $\{f_{pq}\}$ matrix in a single line of code. The next line calls our brand-new two-dimensional FFT function to create the $\{\tilde{f}_{kl}\}$ matrix. We then do something analogous for the Fourier-space indices, i.e., we create two coordinate matrices for the k's and l's, which we use to carry out the division in Eq. (8.137) most simply. There is a slight complication involved here: the element `[0,0]` leads to division with zero, which gives rise to a `RuntimeWarning`. We're not really worried about this, however: this is simply the DC component; our boundary conditions are periodic, as per Eq. (8.129), so we are only able to determine the potential ϕ_{pq} up to an overall constant. Thus, after our division we set this overall offset to obey our chosen normalization. As per our earlier strategy, we then take the inverse DFT in Eq. (8.134) to calculate ϕ_{pq} from $\tilde{\phi}_{kl}$ and then we're done.

The main program calls our function `poisson()`, passing in $n = 128$: as you may

[24] Actually, it returns (x_q, y_p), if you don't employ the optional `indexing` argument. In any case, for us the `xs` and the `ys` are identical, so nothing changes. Try out a simple example if this is not transparent.

poisson.py Code 8.7

```
from fft import fft
import numpy as np

def fft2(A):
     B = A.astype(complex)
     for i, row in enumerate(A):
          B[i,:] = fft(row)
     for j, col in enumerate(B.T):
          B[:,j] = fft(col)
     return B

def inversefft2(A):
     n2 = A.size
     newA = fft2(np.conjugate(A))/n2
     return np.conjugate(newA)

def func(x,y):
     return np.cos(3*x+4*y) - np.cos(5*x-2*y)

def poisson(a,b,n):
     h = (b-a)/(n-1)
     xs = a + np.arange(n)*h
     Xs, Ys = np.meshgrid(xs,xs)

     F = func(Xs, Ys)
     Ftil = fft2(F)

     ks = np.arange(n)
     Kxs, Kys = np.meshgrid(ks,ks)
     Denom = np.cos(2*np.pi*Kxs/n) + np.cos(2*np.pi*Kys/n) - 2
     Phitil = 0.5*Ftil*h**2/Denom
     Phitil[0,0] = 0

     Phi = np.real(inversefft2(Phitil))
     return Phi

if __name__ == '__main__':
     Phi = poisson(0, 2*np.pi, 128); print(Phi)
```

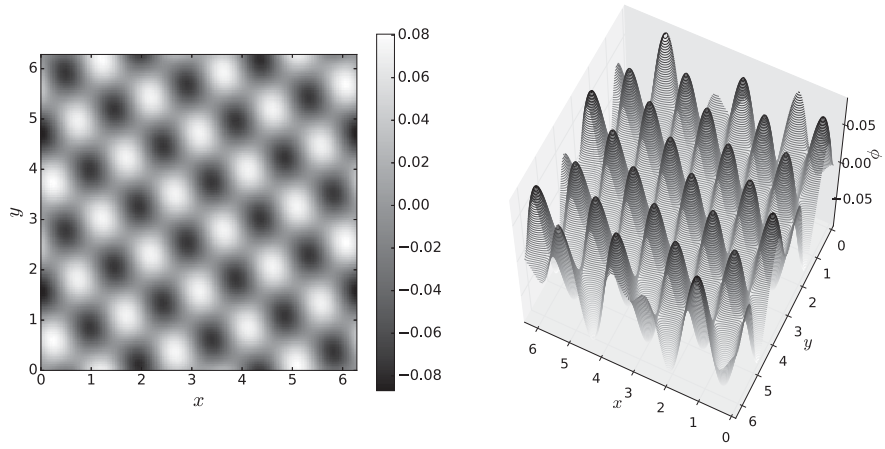


Fig. 8.12 Poisson-equation solution for a two-dimensional periodic boundary-value problem

recall from chapter 6, our implementation of the (one-dimensional) FFT algorithm kept halving, so it relied on n being a power of 2. This routine is here used to carry out the two-dimensional direct and inverse FFT, which in our case gives rise to matrices of dimension 128×128. (You might want to try larger values of n and compare the timing with that resulting from the "slow" DFT implementation.) In Fig. 8.12 we are attempting to visualize the $\{\phi_{pq}\}$, which is our final result. We are showing two different ways of plotting the value of the electric field. Unsurprisingly, the solution is periodic; the coefficients in the arguments of the cosines in the $f(x, y)$ term have led to rapid modulations. In all, we managed to solve a two-dimensional boundary-value problem with a minimum of fuss (and a code that fits on a single page).

8.6 Problems

1. It is standard to assume that a (linear) second-order ODE doesn't have a first-order derivative term. To see why this is legitimate, start with $y''(x) + p(x)y'(x) + q(x) = 0$ and make the substitution:

$$z(x) = y(x)e^{\frac{1}{2}\int p(x)dx} \tag{8.141}$$

   Show that you get the ODE $z''(x) + r(x)z(x) = 0$, which contains no first-order term.
2. In the main text we investigated the stability of the Euler method(s) using the test equation, see Eq. (8.33) and below; here we generalize that discussion. Start from Eq. (8.25), which relates e_{j+1} to e_j. Multiply and divide the term in square brackets with e_j and then interpret the fraction appropriately. One of the terms on the right-hand side should be e_j times a factor that is linear in h; explain what this means.
3. Reproduce Fig. 8.4, augmented to also include results for the explicit midpoint and explicit trapezoid methods, Eq. (8.54) and Eq. (8.57).

4. Implement the backward Euler method for the Riccati equation, Eq (8.71). You could use a root-finder or simply solve the quadratic equation "by hand"; as part of this process, you need to pick one of the two roots (justify your choice).
5. Show that the implicit trapezoid method, Eq. (8.63), has local error $O(h^3)$. To do so, you can re-use the first equality in Eq. (8.44). The main trick involved is that, just like $y'(x_j) = f(x_j, y(x_j))$ holds, we can also take $y'(x_{j+1}) = f(x_{j+1}, y(x_{j+1}))$ and then Taylor expand $y'(x_{j+1})$ around x_j.
6. In the main text we explicitly derived the prescription for second-order Runge–Kutta methods, see section 8.2.2. When we studied the fourth-order Runge–Kutta method, we provided motivation from quadrature and then carried out an explicit derivation for the test equation. We now turn to *third*-order Runge–Kutta methods; to make things manageable, focus on an autonomous ODE, $y' = f(y)$, which is a specific case, but still more general than the single example of the test equation. The prescription is:

$$\begin{aligned} k_0 &= hf(y_j), \qquad k_1 = hf\left(y_j + c_0 k_0\right), \qquad k_2 = hf\left(y_j + c_1 k_0 + c_2 k_1\right) \\ y_{j+1} &= y_j + d_0 k_0 + d_1 k_1 + d_2 k_2 \end{aligned} \tag{8.142}$$

   Your task is, by carrying out a number of Taylor expansions as before, to derive the relations these c and d parameters need to obey.
7. Implement the following (explicit) two-step *Adams–Bashforth method*:

$$y_{j+1} = y_j + \frac{h}{2}\left[3f(x_j, y_j) - f(x_{j-1}, y_{j-1})\right] \tag{8.143}$$

   for the Riccati equation, Eq (8.71). Obviously, you'll need two y values to get going (y_0 and y_1); since a first-order IVP corresponds to only a single input y value (y_0), we'll need to produce an estimate of y_1 before we can start applying this method; use the forward Euler method to do that.
8. Reproduce the two plots shown in Fig. 8.5. Note that you will need to tailor the backward Euler routine to the specific equation you are tackling each time.
9. Tackle the problem in Eq. (8.72) by setting $u(x) = y(x) + e^{-500x}$. As you will discover when you implement this yourself, this trick eliminates RK4's troubles, even though the ODE is unchanged. Do you understand why?
10. Use RK4 to solve the following problem:

$$y'(x) = 100\,(\sin x - y)\,, \qquad y(0) = 0 \tag{8.144}$$

    from $x = 0$ to $x = 8$, with $n = 285$ and $n = 290$ points. Do you understand why the two results are so dramatically different?
11. Generalize Code 8.1 to use Chebyshev points. Then, employ Lagrange interpolation in between the points and produce a new version of Fig. 8.4.
12. Implement global and local adaptive stepping in RK4, to reproduce the panels in Fig. 8.6.

13. The *Runge–Kutta–Fehlberg method* (RK5), with local error $O(h^6)$, follows the steps:

$$\begin{aligned}
k_0 &= hf(x_j, y_j) \\
k_1 &= hf\left(x_j + \frac{h}{4}, y_j + \frac{k_0}{4}\right) \\
k_2 &= hf\left(x_j + \frac{3}{8}h, y_j + \frac{3}{32}k_0 + \frac{9}{32}k_1\right) \\
k_3 &= hf\left(x_j + \frac{12}{13}h, y_j + \frac{1932}{2197}k_0 - \frac{7200}{2197}k_1 + \frac{7296}{2197}k_2\right) \\
k_4 &= hf\left(x_j + h, y_j + \frac{439}{216}k_0 - 8k_1 + \frac{3680}{513}k_2 - \frac{845}{4104}k_3\right) \\
k_5 &= hf\left(x_j + \frac{h}{2}, y_j - \frac{8}{27}k_0 + 2k_1 - \frac{3544}{2565}k_2 + \frac{1859}{4104}k_3 - \frac{11}{40}k_4\right) \\
y_{j+1} &= y_j + \frac{16}{135}k_0 + \frac{6656}{12825}k_2 + \frac{28561}{56430}k_3 - \frac{9}{50}k_4 + \frac{2}{55}k_5
\end{aligned} \tag{8.145}$$

The reason this approach is popular is that one can embed a fourth-order (Runge–Kutta) technique in it (which has local error $O(h^5)$), which we denote by $\tilde{y}_{j+1}$; subtracting the two estimates gives us an expression for the error:

$$y_{j+1} - \tilde{y}_{j+1} = \frac{1}{360}k_0 - \frac{128}{4275}k_2 - \frac{2197}{75240}k_3 + \frac{1}{50}k_4 + \frac{2}{55}k_5 \tag{8.146}$$

This doesn't require the step-halving that we employed for RK4, see Eq. (8.76) and below. Implement both the technique itself and its error estimate programmatically for the Riccati equation, Eq (8.71), and compare with RK4 for fixed $n = 7$.

14. In *molecular dynamics* simulations, the *velocity Verlet algorithm*,[25] tackles the second-order ODE $w'' = f(x, w, w')$ by splitting it into two first-order ODEs, employing the definitions in Eq. (8.84), as usual. However, instead of bundling $y_0(x)$ and $y_1(x)$ into a vector $\mathbf{y}(x)$ and stepping through each in parallel like we did in the main text, the velocity Verlet method employs different steps for each of the two functions:

$$\begin{aligned}
k &= (y_1)_j + \frac{h}{2}f[x_j, (y_0)_j, (y_1)_j] \\
(y_0)_{j+1} &= (y_0)_j + hk, \qquad (y_1)_{j+1} = k + \frac{h}{2}f[x_{j+1}, (y_0)_{j+1}, k]
\end{aligned} \tag{8.147}$$

Implement this explicit method and compare its output with RK4 for the Legendre differential equation, Eq. (8.91), for $n = 6$ and $n = 100$.

15. To solve the time-independent Schrödinger equation, one often employs *Numerov's method*. For $w''(x) = f(x)w(x)$, which is a linear second-order ODE, the Numerov algorithm produces the next value of the function by:

$$w_{j+1} = \frac{\left[2 + \frac{5}{6}h^2 f(x_j)\right] w_j - \left[1 - \frac{1}{12}h^2 f(x_{j-1})\right] w_{j-1}}{1 - \frac{1}{12}h^2 f(x_{j+1})} \tag{8.148}$$

[25] The same idea goes under the name of the *leapfrog method* in the study of partial differential equations.

This discretizes the $w(x)$ directly, i.e., in contradistinction to what we did for other methods, here we do not define $y_0(x)$ and $y_1(x)$. Solve the problem:

$$w''(x) = \frac{5}{1+x^2} w(x), \qquad w(0) = 1, \qquad w'(0) = 0 \tag{8.149}$$

using both Numerov's method and RK4. Observe that Numerov's method is not self-starting, since it requires two w values to get going; however, in adition to $w_0 = w(0)$, you also know $w'(0)$, so you can use it to approximate w_1.

16. Use RK4 to solve the following *Lotka–Volterra equations*, used to describe the populations of a predator and a prey species:

$$y_0'(x) = 0.1y_0(x) - 0.01y_0(x)y_1(x), \qquad y_1'(x) = -0.5y_1(x) + 0.01y_0(x)y_1(x),$$
$$y_0(0) = 60, \qquad y_1(0) = 20 \tag{8.150}$$

Plot $y_0(x)$ and $y_1(x)$, integrating up to $x = 80$.

17. Derive Eq. (8.4) for the problem of the two-dimensional projectile in the presence of air resistance. Implement this programmatically, using `rk4_gen()` and plot the trajectory (i.e., $y(t)$ as a function of $x(t)$). To be concrete, take $k = 1, g = 9.81, x(0) = 1, v_x(0) = 2, y(0) = 5$, and $v_y(0) = 7.808$; integrate from $t = 0$ up to $t = 2.5$.

18. Solve the following IVP using RK4:

$$w''(x) = -w(x) - \frac{1}{x} w'(x), \qquad w(0) = 1, \qquad w'(0) = 0 \tag{8.151}$$

integrating up to $x = 0.5$. Applying `rk4_gen()` gets you in trouble due to the $1/x$ term at $x = 0$. Apply L'Hôpital's rule to that term, getting another contribution of $w''(x)$ at the origin; code up `fs()` so that it takes into account both possibilities. (This is the *Bessel equation* for the *Bessel function* $J_0(x)$, so you may wish to use `scipy.special.jv()`.)

19. First, repeat the analysis in Eq. (8.98) and below, this time for the case of the backward Euler method. Then, apply both the forward and the backward Euler methods for the system of ODEs in Eq. (8.102) integrating up to $x = 0.1$; start with $n = 1000$ to make sure you implemented both techniques correctly, and then repeat the exercise for $n = 6$.

20. Generalize the discussion on stability of the backward Euler method from the previous problem, for the case of an autonomous, possibly nonlinear, system of equations:

$$\mathbf{y}_{j+1} = \mathbf{y}_j + h\mathbf{f}(\mathbf{y}_{j+1}) \tag{8.152}$$

Linearize this system, by Taylor expanding $\mathbf{f}(\mathbf{y}_{j+1})$ around $\mathbf{y}_j$, just like in Eq. (5.76) when introducing Newton's method for root-finding in many dimensions. Formally solve the resulting equation for $\mathbf{y}_{j+1}$ and discuss what condition needs to be satisfied in order for you to do so in a dependable manner.

21. Solve Eq. (8.4) for the problem of the two-dimensional projectile in the presence of air resistance, this time as a BVP, i.e., with the input being $x(0) = 2, v_x(0) = 3, y(0) = 4$, and $y(2.5) = 0$, compute the needed $v_y(0)$. While benchmarking, try out the starting values given in an earlier problem (where you know how large $v_y(0)$ turns out to be).

22. Apply the shooting method to the following fourth-order BVP:

$$w''''(x) = -13w''(x) - 36w(x), \quad w(0) = 0, \quad w'(0) = -3, \quad w(\pi) = 2, \quad w'(\pi) = -9 \tag{8.153}$$

Note that we are missing two pieces of information at the starting point, $w''(0)$ and $w'''(0)$, so you will have to use the multidimensional Newton's method. Specifically, make sure to define one function for the four right-hand sides of our system of ODEs (after you rewrite our fourth-order ODE) and another function for the two conditions we wish to satisfy ($w(\pi) - 2 = 0$ and $w'(\pi) + 9 = 0$).

23. We will now see how to handle *Neumann boundary conditions* in the matrix approach to the BVP. Specifically, take the following problem:

$$\begin{gathered} w''(x) = -\frac{30}{1-x^2}w(x) + \frac{2x}{1-x^2}w'(x) \\ w'(0.05) = 1.80962109375, \qquad w(0.49) = 0.1117705085875 \end{gathered} \tag{8.154}$$

which is a variation on Eq. (8.106). Our finite-difference scheme leads to equations like those in Eq. (8.111), with the first one being replaced by:

$$\frac{w_1 - w_{-1}}{2h} = 1.80962109375 \tag{8.155}$$

Here the solution is *not* required to be periodic, so you *cannot* assume $w_{-1} = w_{n-1}$ holds, as in Eq. (8.124): w_{-1} is simply outside the solution domain. Apply the discretized ODE, Eq. (8.110), for the case of $j = 0$; combining this with Eq. (8.155) allows you to eliminate w_{-1}. Implement the solution to this BVP programmatically.

24. After introducing the matrix approach to the BVP in Eq. (8.108), we discussed the error stemming from the finite-difference approximations employed. If you consider the problem more carefully, you will realize that the error in approximating the derivatives is not necessarily the same as the error in approximating $w(x_j)$ by w_j (which quantifies how well or how poorly we are satisfying the ODE we are trying to solve). Expand $w(x_{j+1})$ and $w(x_{j-1})$, both around x_j, to explicitly investigate how good an approximation w_j is, for the case of a linear BVP.

25. Solve the following nonlinear BVP:

$$w''(x) = -e^{w(x)}, \qquad w(0) = 1, \qquad w(1) = 0 \tag{8.156}$$

using both the shooting method and the matrix approach. Note that you'll need a root-finder for both cases (one-dimensional and multidimensional, respectively). Try out different initial guesses to find *two* solutions to this problem. For the matrix approach, have your initial guess vector be either zero or $20x - 20x^2$; you could evaluate the Jacobian numerically (as in `multi_newton.py`) or analytically.

26. Experiment with the shooting method for an eigenvalue problem. Specifically, reproduce Fig. 8.8 which shows that it is the eigenvalue, and not the starting derivative, which allows us to satisfy the boundary condition.

27. Reproduce our plots of the Mathieu functions in Fig. 8.9 using `evp_shoot.py` for the left panel and `evp_matrix.py` for the right panel.

28. Use the shooting method for the quantum-mechanical eigenvalue problem of the *infinite square well* (which is different from the periodic box we encountered earlier). Specifically, take the potential to be zero for $-a < x < a$ and infinite at the boundaries; compute the first six eigenvalues and compare with:

$$E_n = \frac{\hbar^2 \pi^2}{8ma^2} n^2 \tag{8.157}$$

which you learned in a course on quantum mechanics. Take $\hbar^2/m = 1$ and $a = 0.5$.

29. Use Code 8.6 to tackle the *quantum harmonic oscillator*, for which the "potential" term in Eq. (8.116) is $0.5qx^2$ instead of $2q \cos 2x$. Write separate functions for two finite-difference discretization schemes: first, Eq. (8.121) which has error $O(h^2)$ and, second, the approximation which has error $O(h^4)$. Feel free to use `numpy.linalg.eigvals()`. The eigenfunctions are "periodic" in that they are expected to die off at large distances. Compare the answer to the (analytically known) eigenvalues of Eq. (3.71).
30. For the first six eigenvalues of the quantum harmonic oscillator of the previous problem, apply *Richardson extrapolation*: for a given finite-difference approximation to the second derivative, say, that of Eq. (8.121), carry out a calculation for step size h and another one for $h/2$ and use Eq. (3.47); compare with the "unextrapolated" values.
31. If you take $q = 0$ in Code 8.6, i.e., `evp_matrix.py`, then you get the non-interacting particle in a box (of length 2π), see section 3.5.1. From Eq. (3.82) the energy is analytically known to be (proportional to) n^2; this means that (except for the 0th eigenvalue) the energies will come in pairs, e.g., $n = \pm 3$. For the first 50 distinct eigenvalues, plot: (a) the exact eigenvalue, and (b) the numerical answer for matrix-approach discretizations using $n = 100, 200, 300, 400, 500$ vs the cardinal number of the eigenvalue; feel free to use `numpy.linalg.eigvals()`. Do you notice a pattern?
32. In quantum mechanics, the Mathieu equation takes the form:

$$-\frac{\hbar^2}{2m}\psi''(x) + 2q \cos 2x\, \psi(x) = E\psi(x) \tag{8.158}$$

In the spirit of the Project in section 7.8, we can view the Hamiltonian on the left-hand side as $\hat{H} = \hat{H}_0 + \hat{H}_1$. We will now see how to tackle this problem in perturbation theory. In the previous problem we took $q \to 0$: this leads to the non-interacting particle-in-box, with the eigenfunctions and eigenenergies being given by Eq. (3.80) and Eq. (3.82), respectively; let us denote the state vector of the non-interacting problem by $|n^{(0)}\rangle$ and the corresponding energy by $E_n^{(0)}$. Show that the first-order correction to the energy, $E_n^{(1)} = \langle n^{(0)}|\hat{H}_1|n^{(0)}\rangle$, vanishes. Then, derive a simple analytical expression for the second-order correction to the energy, starting from:

$$E_n^{(2)} = \sum_{k \neq n} \frac{|\langle n^{(0)}|\hat{H}_1|k^{(0)}\rangle|^2}{E_n^{(0)} - E_k^{(0)}} \tag{8.159}$$

33. Compute the first six eigenvalues of the Mathieu equation (for $q = 1.5$ with $n = 150$) using a higher-order approximation for the second derivative. That is, instead of Eq. (8.121) which has error $O(h^2)$, use a finite-difference approximation which has error $O(h^8)$.

34. In the main text (implicitly) and in earlier problems (explicitly), we have been treating the Schrödinger equation as a differential equation, working in the position basis. We now learn that it can be cast and solved as an *integral equation*, by working in the momentum basis. First, show that, by using the resolution of the identity in one dimension, $\int dk|k\rangle\langle k|$, the Schrödinger equation in Eq. (4.342) takes the form:

$$\frac{k^2}{2m}\psi(k) + \int dk' V(k,k')\psi(k') = E\psi(k) \tag{8.160}$$

Use numerical quadrature and show that this equation is equivalent to:

$$\sum_j H_{ij}\psi_j = E\psi_j \tag{8.161}$$

This is the eigenvalue problem $\mathbf{H}\boldsymbol{\psi} = E\boldsymbol{\psi}$, see Eq. (4.262). Finally, compute the ground-state energy for $V(k,k') = -e^{-(k-k')^2/4}/(2\sqrt{\pi})$; use 100 Gauss–Legendre points from -10 to $+10$. While benchmarking, use the matrix approach to also solve the corresponding differential equation in coordinate space, for $V(x) = -e^{-x^2}$.

35. Compute $\phi(x,y)$ for the problem of Eq. (8.129), allowing for different discretizations in the x and y coordinates. Specifically, reproduce Fig. 8.12 for $n_x = 64$ and $n_y = 128$.

36. In a course on electromagnetism you've probably seen how to solve Poisson's equation via the use of *Green's functions*; for the two-dimensional case this takes the form:

$$\phi(x,y) = \int \frac{f(x',y')}{|\mathbf{r}-\mathbf{r}'|} d^2r' \tag{8.162}$$

Compute the electric potential for the problem of Eq. (8.129) by carrying out this two-dimensional integral numerically, say, via Gauss–Legendre quadrature; note that you need to compute such an integral *for each* value of x and y. As if that wasn't bad enough, you now have to worry about the denominator value(s) being close to zero.

37. Tackle the problem of Eq. (8.129) by setting up Eq. (8.133) as a linear algebra problem and solving it. Take $n = 4$ and assume that $\phi = 7$ at the boundary. This leads to nine equations for the nine unknown $\phi_{p,q}$'s, for the interior points. Start from $p = q = 1$: some of the terms in Eq. (8.133) are known from the boundary condition, so they can be moved to the right-hand side. Explicitly write out the other eight equations, grouping everything into the form $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{x}$ bundles the nine interior $\phi_{p,q}$'s column-wise: that means it contains $\phi_{1,1}, \phi_{2,1}, \phi_{3,1}, \phi_{1,2}, \phi_{2,2}, \ldots$.

38. Solve a problem that is similar to Eq. (8.129), but this time for a driving term which is $f(x,y) = ce_1(3x+4y) - ce_1(5x-2y)$, where ce_1 is the second even Mathieu function for the case of $q = 1.5$; make sure to normalize this Mathieu function in such a way that $ce_1(0) = 1$. You want to be able to evaluate the right-hand side of Poisson's equation for any possible argument, so implement ce_1 as a combination of an eigenvalue-problem (to produce the function on a grid) and an interpolation via FFT (to allow you to compute its value for points off the grid).

> Someone had been reading aloud for a long time and was about to finish; pointing to the blank space on the scroll, Diogenes cried: "Cheer up, fellows, land is in sight!".
>
> Diogenes Laërtius (on Diogenes the Cynic)