Part Two

Numerical Models

By now it should be evident that the fraction of differential equations that can be solved exactly with a sane amount of effort is quite small, and that once we stray too far from linearity special techniques are needed for there to be any hope of writing down a closed-form solution. In this second part of the book we will turn to the appealing alternative: numerical solutions. Although the widespread access to fast computers has perhaps led to an over-reliance on numerical answers when there are other possibilities, and a corresponding false sense of security about the possibility of serious numerical problems or errors, it is now possible without too much trouble to find solutions to most equations that are routinely encountered.

An essential issue is when the result of a numerical calculation can be trusted. It's (almost) always possible to produce some kind of number, but it's much harder to produce a meaningful number. One good rule of thumb is that the result should not depend on the algorithm parameters (for example, decreasing a step size should give the same answer), otherwise the result provides information about the algorithm rather than the underlying problem. Another crucial sanity test is to check the algorithm on a problem with a known exact solution to make sure that it agrees. Even so, there's still a chance that a subtle difference between the algorithm and the underlying problem can lead to fundamentally different results.

In the commonly-used IEEE floating point standard [IEEE, 1985], a 32-bit *single precision* floating-point number has a 24 bit *mantissa* (fractional part), corresponding to about 7 decimal digits, and a 64-bit *double precision* number has 53 bits or about 16 digits. It is common to assume that roundoff errors beyond this act like a random noise term, but it is possible that the errors are strongly correlated rather than random [Sauer *et al*., 1997]. *Interval arithmetic* is an alternative approach that recognizes that a floating-point number actually represents a range of real numbers rather than a single value [Hickey *et al*., 2001]. Mathematical functions are redefined to operate on these intervals, returning the limits of possible values. This doesn't cure errors from limited-precision calculations, but it does help identify them and guide the development of algorithms to reduce them.

A principle that is so important for numerical methods that it doesn't even have a name (and is frequently ignored) is that if you know your system has some conservation laws then you should choose variables that automatically conserve them. Otherwise, your conservation laws will be ignored by the solution, or will need to be explicitly enforced. Not only does this make your solution more accurate and reduce the computational effort, it can help tame numerical instabilities that might otherwise occur.

More generally, remember that numerical and analytical solutions should not be viewed as exclusive alternatives, but rather as complementary approaches. Many important ideas have come at this interface, such as *solitons*. These nonlinear nondispersive waves first appeared as unusual stable structures in numerical simulations of plasmas; this observation motivated and then guided the successful search for corresponding analytical solutions [Miles, 1981; Zabusky, 1981].

There is a continuum between numerical and analytical solutions, and a trade-off between the need for computer power and mathematical insight. The more that an algorithm takes advantage of knowledge about an equation, the less work that the computer need do. However, given a fast computer and a hard problem a less clever algorithm may be preferable. In the early days of numerical mathematics, algorithms were implemented by people called *calculators* (sometimes large rooms full of them) using slide rules, ta-

bles, or arithmetic machines. This put a large premium on reducing the number of steps needed, and hence on developing algorithms that maximize step sizes and minimize the number of needed function evaluations. As desktop workstations now begin to exceed the speed of recent supercomputers, less efficient algorithms that have other desirable properties (such as simplicity or reliability) can be used. We will cover the most important algorithms that are straightforward to understand and implement; for these as well as everything else related to numerical methods [Press *et al*., 2007] is a great starting point for more information. Another interesting reference is [Acton, 1990], a revised edition of a beautiful book (with a sneaky cover) that has had a large impact on many people working in the field of numerical analysis.

Some caution is needed in delving more deeply into the literature on numerical analysis. Often the algorithms that are most useful in practice are the ones that are least amenable to proving rigorous results about their error and convergence properties, and hence are less well represented. It's necessary to condition what is known against a usually unstated prior of what kinds of problems are likely to be encountered.

To a computational complexity theorist, the most important distinction in deciding if a problem is tractable is the difference between those that can be solved in a number of steps that is polynomial in the size of the problem, such as sorting, and those that cannot (or at least are believed to grow faster than polynomial, such as factoring) [Lewis & Papadimitriou, 1981]. For example, the *number field sieve* [Lenstra & Lenstra, Jr., 1993] for finding the prime factors of a number N requires $\mathcal{O}(e^{1.9(\ln N)^{1/3}(\ln \ln N)^{2/3}})$ steps; using it, a 512-bit number was factored in 5 months using 8000 MIPS (*Millions of Instructions Per Second*) years of distributed parallel computer time [Atkins *et al*., 1995]. But if you're fortunate enough to own a quantum computer the problem could be done in $\mathcal{O}\left((\ln N)^2\right)$ steps [Shor, 1997], which for a 512-bit number is just $\mathcal{O}(10^5)$ instructions. It can also be possible to *relax* an exponential problem to find a related one that can be solved in polynomial time (Chapter 15).

However, to a working numerical analyst, the relevant distinction is really between $\mathcal{O}(N^2)$ algorithms and faster ones, which in practice is the distinction between what is feasible and what is not for nontrivial problems. Naively, a Fourier transform requires $\mathcal{O}(N^2)$ steps because it needs a matrix multiplication, but by taking advantage of the structure in the calculation the *Fast Fourier Transform* algorithm reduces it to $\mathcal{O}(N \log N)$ (Section 13.2). While this difference might not seem as remakable as the difference between exponential and polynomial time, in practice it is profound. For example, for $N = 10$, an N^2 algorithm requires 100 steps, and an $N \log N$ algorithm requires 33 steps, not much difference. But for $N = 10^9$, an N^2 algorithm requires 10^{18} steps (10^5 days at 100 MHz), while $N \log N$ requires 3×10^{10} steps (300 seconds), quite a difference indeed! This issue of the scaling of an algorithm with problem size will recur throughout the coming chapters, and is one of the most important lessons in all of numerical analysis. There are endless of examples of promising new algorithms that do not survive the scaling up to nontrivial problems.

7 Finite Differences: Ordinary Differential Equations

7.1 NUMERICAL APPROXIMATIONS

This chapter will consider the problem of finding the numerical solution to the first-order (usually nonlinear) differential equation

$$\frac{dy}{dx} = f(x, y) \quad . \tag{7.1}$$

Because of the presence of y on the right hand side we can't simply integrate $f(x)\,dx$; we'll need some kind of iterative procedure to calculate a new value of y and use it to evaluate f. Fortunately there are techniques for solving ODEs that are relatively straightforward to implement and use, and that are broadly applicable. If $\dot{y} = dy/dx$ also appears on the right hand side, $f(x, y, \dot{y}) = 0$, this becomes a *differential-algebraic equation* (*DAE*), a still harder problem that usually requires more complex algorithms matched to the problem [Brenan *et al*., 1996].

The restriction to first-order equations actually isn't much of a restriction at all. The algorithms for solving a single first-order equation will immediately generalize to systems of equations

$$\begin{aligned}
\frac{dy_1}{dx} &= f_1(x, y_1, \ldots, y_N) \\
\frac{dy_2}{dx} &= f_2(x, y_1, \ldots, y_N) \\
&\vdots \\
\frac{dy_N}{dx} &= f_N(x, y_1, \ldots, y_N) \quad .
\end{aligned} \tag{7.2}$$

And, since a higher-order differential equation of the form

$$\frac{d^N y}{dx^N} = f\left(x, y, \frac{dy}{dx}, \ldots, \frac{d^{N-1}y}{dx^{N-1}}\right) \tag{7.3}$$

can be written as a system of first-order equations (equation 3.20)

$$\begin{aligned}
y^{(1)} \equiv \frac{dy}{dx} \quad \cdots \quad y^{(N-1)} &\equiv \frac{d^{N-1}y}{dx^{N-1}} \\
\frac{dy^{(1)}}{dx} &= y^{(2)} \\
&\vdots
\end{aligned}$$

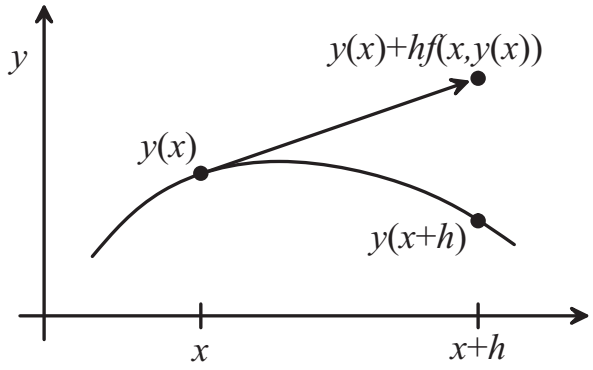

Figure 7.1. An Euler method.

$$\begin{aligned} \frac{dy^{(N-1)}}{dx} &= y^{(N)} \\ \frac{dy^{(N)}}{dx} &= f(x, y^{(1)}, \ldots, y^{(N-1)}) \quad , \end{aligned} \tag{7.4}$$

we will also be able to solve higher-order equations.

We would like to find an approximate formula to relate $y(x+h)$ to $y(x)$ for some small step h. An obvious way to do this is through the Taylor expansion of y with respect to h:

$$\begin{aligned} y(x+h) &= y(x) + h\frac{dy}{dx}\bigg|_x + \frac{h^2}{2}\frac{d^2y}{dx^2}\bigg|_x + \mathcal{O}(h^3) \\ &= y(x) + hf(x, y(x)) + \frac{h^2}{2}\frac{d}{dx}f(x, y(x)) + \mathcal{O}(h^3) \\ &= y(x) + hf(x, y(x)) + \frac{h^2}{2}\left[\frac{\partial f}{\partial x} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial x}\right] + \mathcal{O}(h^3) \\ &= y(x) + hf(x, y(x)) + \frac{h^2}{2}\left[\frac{\partial f}{\partial x} + f\frac{\partial f}{\partial y}\right] + \mathcal{O}(h^3) \quad . \end{aligned} \tag{7.5}$$

An approximation scheme must match these terms to agree with the Taylor expansion up to the desired order.

The first two terms of the expansion

$$y(x+h) = y(x) + hf(x, y(x)) \tag{7.6}$$

can be used to find $y(x+h)$ given $y(x)$, and this step can then be repeated to find $y(x+2h) = y(x+h) + hf(x+h, y(x+h))$, and so forth. This is the simplest algorithm for solving differential equations, called *Euler's method* (shown in Figure 7.1). It is simple to understand and simple to program; its clarity is matched only by its dreadful performance. At each step the error is $\mathcal{O}(h^2)$ (the lowest-order term where the Euler method differs from the Taylor expansion of $y(x)$ is the h^2 term), and so a very small step size is needed for a reasonably accurate solution. Even worse, the errors can accumulate so rapidly that the numerical solution actually becomes unstable and blows up. Consider

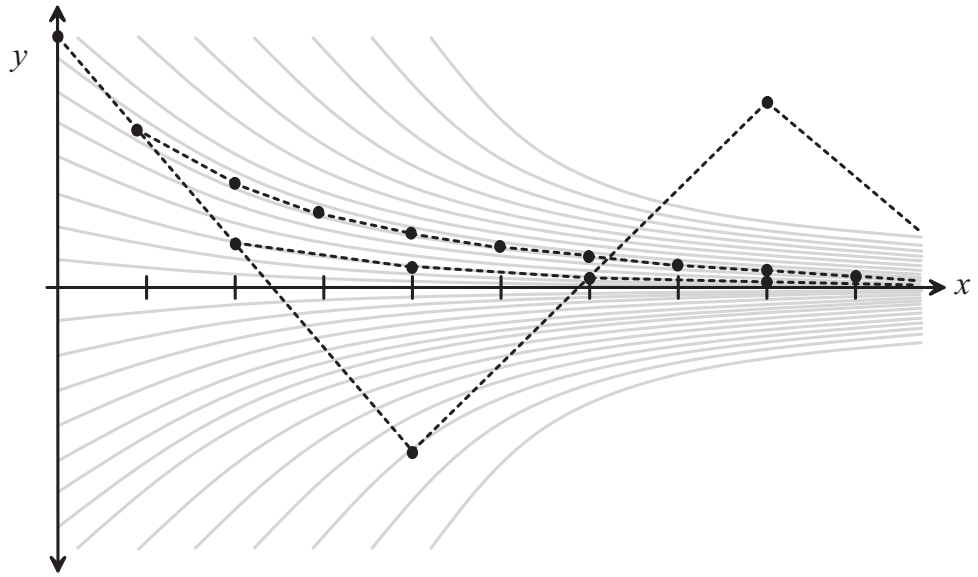

Figure 7.2. Origin of oscillation in the Euler method. The gray lines show the family of solutions of $dy/dx = Ay$, and the dotted lines show the numerical solution for various step sizes.

the simple differential equation

$$\frac{dy}{dx} = Ay \quad . \tag{7.7}$$

The exact solution is $y(x) = e^{Ax}$; if $A < 0$ then $\lim_{x\to\infty} y(x) = 0$. The Euler approximation for this equation is

$$\begin{aligned} y(x+h) &= y(x) + hAy(x) \\ &= (1+hA)y(x) \quad . \end{aligned} \tag{7.8}$$

This is a first-order difference equation, with a solution equal to $y(x) = (1+hA)^{x/h}y(0)$. If $A > 0$ this solution diverges, as it should. If $0 > hA > -1$, then the solution properly decays to zero. But look what happens as hA becomes even more negative. If $-1 > hA > -2$, the magnitude of the solution still decays, but it has picked up an oscillation that is not in the original equation. Even worse, if $-2 > hA$, then the magnitude of the solution will diverge! We've solved an equation, but it now has nothing to do with our original differential equation. This is an example of *von Neumann stability analysis*, developed in the next chapter.

Figure 7.2 shows why the Euler method is such a poor approximation. Since the derivative is evaluated only at the beginning of the interval, if an overly ambitious step size is chosen the extrapolation can overshoot so far that the solution changes sign. A natural improvement is to use an Euler step in order to estimate the slope in the middle of the interval, and then use this slope to update y:

$$y(x+h) = y(x) + hf\left[x + \frac{h}{2},\ y(x) + \frac{h}{2}f(x, y(x))\right] \quad . \tag{7.9}$$

The error made by this approximation can be found by doing a Taylor expansion as a function of h:

$$\begin{aligned} & f\left[x+\frac{h}{2},\ y(x)+\frac{h}{2}f(x,y(x))\right] \\ &= f(x,y(x))+h\frac{d}{dh}f\left[x+\frac{h}{2},\ y(x)+\frac{h}{2}f(x,y(x))\right]_{h=0}+\mathcal{O}(h^2) \\ &= f(x,y(x))+h\left[\frac{1}{2}\frac{\partial f}{\partial x}+\frac{1}{2}f(x,y(x))\frac{\partial f}{\partial y}\right]+\mathcal{O}(h^2) \end{aligned} \tag{7.10}$$

and so equation (7.9) becomes

$$y(x+h)=y(x)+hf(x,y(x))+\frac{h^2}{2}\left[\frac{\partial f}{\partial x}+f\frac{\partial f}{\partial y}\right]+\mathcal{O}(h^3) \quad . \tag{7.11}$$

Comparing this with equation (7.5), we see that this is exactly the expansion of the solution of the differential equation up to second order. This is called the *second-order Runge–Kutta* or the *midpoint* method. We have found a way to evaluate the function that gives us an answer that is correct to second order, but that does not require explicitly working out the Taylor expansion. Problem 7.2 shows the benefit of improving the approximation order.

7.2 RUNGE–KUTTA METHODS

This procedure can be carried out to use more function evaluations to match higher-order terms in the Taylor expansion. The derivation rapidly becomes very tedious, and there is no unique solution for a given order, but by far the most common approximation is the *fourth-order Runge–Kutta* approximation

$$\begin{aligned} k_1 &= hf(x,y(x)) \\ k_2 &= hf\left(x+\frac{h}{2},\ y(x)+\frac{k_1}{2}\right) \\ k_3 &= hf\left(x+\frac{h}{2},\ y(x)+\frac{k_2}{2}\right) \\ k_4 &= hf(x+h,\ y(x)+k_3) \\ y(x+h) &= y(x)+\frac{k_1}{6}+\frac{k_2}{3}+\frac{k_3}{3}+\frac{k_4}{6}+\mathcal{O}(h^5) \quad . \end{aligned} \tag{7.12}$$

In the midpoint method we improved the accuracy by evaluating the function in the middle of the interval. The fourth-order Runge–Kutta formula improves on that by using two evaluations in the middle of the interval, and one at the end of the interval, in order to make the solution correct out to the fourth-order term in the Taylor series. For a system of equations, this becomes

$$\begin{aligned} k_{1,i} &= hf_i(x,y_1,\ldots,y_N) \\ k_{2,i} &= hf_i\left(x+\frac{h}{2},\ y_1+\frac{k_{1,1}}{2},\ldots,y_N+\frac{k_{1,N}}{2}\right) \\ k_{3,i} &= hf_i\left(x+\frac{h}{2},\ y_1+\frac{k_{2,1}}{2},\ldots,y_N+\frac{k_{2,N}}{2}\right) \end{aligned}$$

$$k_{4,i} = hf_i(x + h,\ y_1 + k_{3,1}, \ldots, y_N + k_{3,N})$$
$$y_i(x + h) = y_i(x) + \frac{k_{1,i}}{6} + \frac{k_{2,i}}{3} + \frac{k_{3,i}}{3} + \frac{k_{4,i}}{6} + \mathcal{O}(h^5) \quad . \tag{7.13}$$

The compactness of this fourth-order formula provides a nice compromise between implementation and execution effort. Although still higher-order approximations are possible, the next section will look at smarter ways to improve the approximation error.

For a given problem, how is the step size h chosen? Obviously, it must reflect the desired accuracy of the final answer. But, how can we estimate the accuracy if we don't know the answer? The simplest solution is to keep reducing the step size until the solution does not change within the desired tolerance. We can be more clever than that, but this is always a good sanity check: the result of a numerical calculation should not depend on the algorithm parameters.

A more intelligent approach is to consider how the approximation error depends on the step size. In a fourth-order method, the deviation between the approximate value found in a single full step $y(x + h)$ and the correct value $y_{\text{true}}(x + h)$,

$$y(x + h) - y_{\text{true}}(x + h) = h^5\varphi(x) + \mathcal{O}(h^6) \quad , \tag{7.14}$$

consists of h^5 times a quantity φ which is approximately constant over the interval (to order h^5; this is the definition of the order of the error). If, instead of a single step h, two smaller steps of $h/2$ are made, the error after the first half-step is

$$y(x + h/2) - y_{\text{true}}(x + h/2) = \left(\frac{h}{2}\right)^5 \varphi(x) + \mathcal{O}(h^6) \tag{7.15}$$

and then after the second half-step it is approximately

$$y(x + h/2 + h/2) - y_{\text{true}}(x + h) = 2\left(\frac{h}{2}\right)^5 \varphi(x) + \mathcal{O}(h^6) \tag{7.16}$$

(it is conventional to assume that the errors at each step add, although in fact this is the worst case and the combined errors for some problems might conspire to be better than that). Therefore, the difference between $y(x + h)$ calculated in a full step of h and in two half-steps of $h/2$ is

$$y(x + h) - y(x + h/2 + h/2) = h^5\varphi(x) - 2\left(\frac{h}{2}\right)^5 \varphi(x) + \mathcal{O}(h^6)$$
$$\approx h^5\varphi \quad . \tag{7.17}$$

The difference between a single step and two half-steps provides an estimate of the local error in the step. Such an error estimate can be used to guide an automatic stepper routine: after making a full step and comparing the result to that from two half-steps, if the error is larger than an upper threshold then the step size is decreased, and if the error is smaller than a lower threshold then it is increased. This allows the program to move to large steps if the function is smooth and featureless, and then to drop down to small steps in regions where the function is complicated.

Note that the local error estimates cannot trivially be added up to get a global error estimate at the end of the calculation. Although it is common to assume that the local errors can be combined as uncorrelated random variables, there are many cases where

they are very correlated and lead the solution away from the correct answer (for example, in the context of solving chaotic equations see [Dawson *et al*., 1994]). And remember that there's a hardware limit to the error that can be achieved; single-precision floating-point numbers typically have about six significant digits, and double-precision numbers have twelve digits.

Having done the extra work needed to make the local error estimate, we can also improve our approximation by combining the results. Equations (7.14) and (7.16) are two equations in the two unknowns y_{true} and φ (ignoring terms of $\mathcal{O}(h^6)$), which can easily be solved to find

$$y_{\text{true}}(x+h) = y(x+h/2+h/2) + \frac{y(x+h/2+h/2) - y(x+h)}{15} + \mathcal{O}(h^6) \quad . \tag{7.18}$$

After checking the error on the fourth-order method by making a full step and two half-steps, this lets the error be reduced by making use of the two estimates for $y(x+h)$ (although to this order we cannot estimate the error in the improved approximation).

A refinement on this step-doubling method due to Fehlberg [Press *et al*., 2007] uses six function evaluations to give a fifth-order Runge–Kutta approximation, and a different combination of the same six values for a fourth-order approximation. Although the functional form is more complex than the standard fourth-order method, this permits an error estimate to be made without needing any extra function evaluations, a desirable trade-off if the function evaluations are computationally costly.

The details of implementing an adaptive interval updating scheme depend on the nature of the equation being integrated. If there are rough regions expected it is crucial to throw away a step with a large error and try again with a smaller step; if the answer is expected to be smooth and execution time is a problem, the point can be saved and the reduced step applied to the following interval. Similarly, small changes in the step size help the routine fine-tune its step, but large changes are needed if the solution varies enormously. If the factors used to increase and decrease the step are incommensurate then it is possible to reach any step size, otherwise the step size will be limited to a rational subset (such as powers of 2). Most generally, adjusting the step size to meet a target local error can be viewed as a control theory problem (Chapter 21).

7.3 BEYOND RUNGE–KUTTA

The combination of a fourth-order Runge–Kutta solver with an adaptive interval stepper is easy to program, easy to use, and can handle most any reasonably well-behaved problem. For this reason it is a workhorse for solving differential equations. This section introduces two important alternatives. At best, they can find more accurate solutions with larger steps and fewer function evaluations, but they are also fussier and can fail catastrophically. Runge–Kutta is always a good starting point; these fancier algorithms should be considered if the execution time or accuracy need to be improved.

The first, *predictor-corrector* methods, start by recognizing that a step in solving the first-order differential equation

$$\frac{dy}{dx} = f(x, y(x)) \tag{7.19}$$

can formally be written as an intergral over an interval h

$$y(x+h) = y(x) + \int_x^{x+h} f(x, y(x))\, dx \quad . \tag{7.20}$$

The problem with this integral is that to evaluate it we need to know $y(x)$, but that is what we're trying to solve for in the first place. All is not lost, however: we do know the history of $f(x, y(x))$ before the interval that we are trying to step over. We ignored this history in the Runge–Kutta methods and just used values in the interval, but if the function is not varying too wildly we can do better and extrapolate over the interval. A common way to do the extrapolation is to assume a polynomial form for f (Chapter 14 will look in detail at other ways to approximate functions). For example, for a third-order method, we assume that locally

$$f(x, y(x)) = a + bx + cx^2 \quad . \tag{7.21}$$

This can easily be integrated:

$$\int_x^{x+h} f(x, y(x))\, dx = ah + bxh + \frac{1}{2}bh^2 + cx^2h + cxh^2 + \frac{1}{3}ch^3 \quad . \tag{7.22}$$

Although it's possible to fit a polynomial at each step to determine the coefficients ($a, b,$ and c here), we can get the same answer by judicious function evaluations. If we cleverly guess that we can write the integral as a sum of past values of the function, weighted by unknown coefficients (α, β, γ),

$$\int_x^{x+h} f(x, y(x))\, dx \stackrel{?}{=} h\{\alpha f[x, y(x)] + \beta f[x-h, y(x-h)] + \gamma f[x-2h, y(x-2h)]\} \quad , \tag{7.23}$$

then plugging in equation (7.21) on the right hand side shows that

$$\int_x^{x+h} f(x, y(x))\, dx = ah(\alpha+\beta+\gamma) + bxh(\alpha+\beta+\gamma) + bh^2(-\beta-2\gamma) + cx^2h(\alpha+\beta+\gamma) + cxh^2(-2\beta-4\gamma) + ch^3(\beta+4\gamma) \quad . \tag{7.24}$$

Equation (7.24) will agree with equation (7.22) if

$$\begin{aligned} \alpha + \beta + \gamma &= 1 \\ -\beta - 2\gamma &= \frac{1}{2} \\ \beta + 4\gamma &= \frac{1}{3} \end{aligned} \quad . \tag{7.25}$$

These equations are easily solved to find $\alpha = 23/12, \beta = -4/3$, and $\gamma = 5/12$, or

$$y_p(x+h) = y(x) + \frac{h}{12}\{23f[x, y(x)] - 16f[x-h, y(x-h)] + 5f[x-2h, y(x-2h)]\} \quad . \tag{7.26}$$

This gives an estimate of $y(x+h)$ based on extrapolating the history of f; for this reason it is called a *predictor* step. Doing an integral with function evaluations like this is an example of *numerical quadrature*.

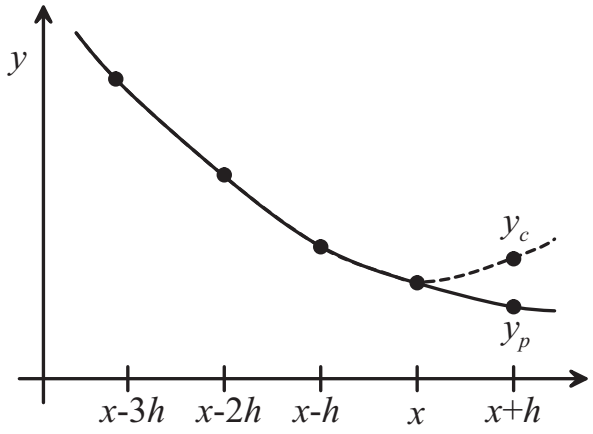

Figure 7.3. A predictor-corrector method.

Although we could use the predicted $y(x+h)$ as the input to a new predictor step, that would try to solve the differential equation by repeated polynomial extrapolations (a bad idea). But we can reapply the differential equation to find an improved estimate based on the prediction; this is called a *corrector* step (Figure 7.3). Since we have an estimate of y at the end of the interval, for the corrector we can look for an implicit numerical quadrature formula that uses it. For the third-order example, we want

$$\int_x^{x+h} f(x, y(x))\, dx \stackrel{?}{=} h\{\alpha' f[x+h, y(x+h)] + \beta' f[x, y(x)] + \gamma' f[x-h, y(x-h)]\} \quad . \tag{7.27}$$

Repeating the preceeding calculation gives $\alpha' = 5/12$, $\beta' = 2/3$, and $\gamma' = -1/12$, or

$$y_c(x+h) = y(x) + \frac{h}{12}\{5f[x+h, y(x+h)] + 8f[x, y(x)] - f[x-h, y(x-h)]\} \quad . \tag{7.28}$$

The result from the corrector step can be used in a new predictor step, which is then corrected, and so forth. Getting this iteration going will require a set of starting values, which can be provided by a self-starting method such as Runge–Kutta. And the difference between the predictor and corrector steps provides a local error estimate. This is an example of an *Adams–Bashforth–Moulton* method; it's common to use the fourth-order form,

$$y_p(x+h) = y(x) + \frac{h}{24}\{55f[x, y(x)] - 59f[x-h, y(x-h)] + 37f[x-2h, y(x-2h)] - 9f[x-3h, y(x-3h)]\} \tag{7.29}$$

and

$$y_c(x+h) = y(x) + \frac{h}{24}\{9f[x+h, y_p(x+h)] + 19f[x, y(x)] - 5f[x-h, y(x-h)] + f[x-2h, y(x-2h)]\} \quad . \tag{7.30}$$

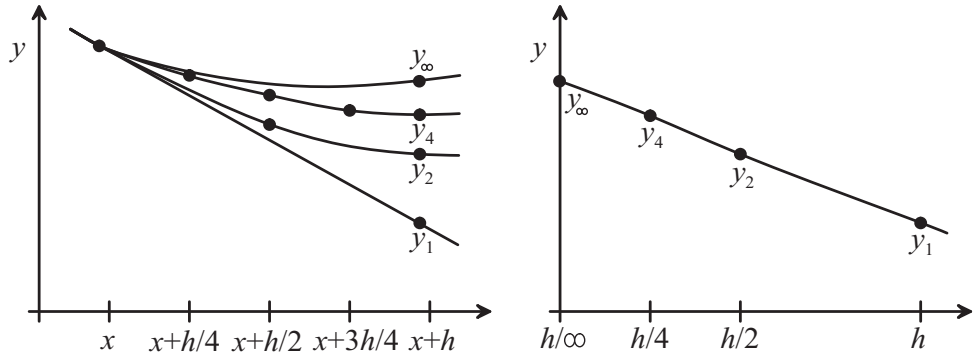

Figure 7.4. Richardson extrapolation.

When polynomial extrapolation is justified, a predictor-corrector routine can make significantly longer steps than a Runge–Kutta routine because it doesn't ignore points that have already been calculated, but it will fail around singularities or discontinuities that are poorly fit by a polynomial.

The idea of extrapolation, plus the step-doubling routine that we used for Runge–Kutta error control, hints at the final numerical method: *Richardson extrapolation*. We saw that two steps of $h/2$ give a smaller final error than one step of h. Four steps of $h/4$ give a smaller error still, and ∞ steps of h/∞ would be even better (exact, in fact). Although it's not very practical to plan on taking infinitely many steps, the sequence leading up to it can give us insight into the infinite limit. The idea is to calculate the value at the end of the interval many times with successively finer steps, and then fit a function to extrapolate to the magical limit of an infinitely small step size (Figure 7.4). The *Bulirsch–Stoer* method uses polynomials or ratios of polynomials to do the extrapolation. Setting this up requires a more complex algorithm with many more internal parameters than Runge–Kutta uses, but in return a *much* larger step size can be used if the solution is not too complex. Predictor-corrector methods are older and better studied than Richardson extrapolation methods, but it is reasonable to believe that it is easier to predict the convergence of a sequence rather than the extrapolation of a complicated function and so extrapolation methods are becoming more common.

We have so far assumed that the differential equation being solved is reasonably well behaved. A particularly nasty source of problems is *stiff* differential equations, which typically arise when a problem has vastly different time or length scales. Consider the following example:

$$\frac{d^2y}{dx^2} - 10^6 y = 0 \quad . \tag{7.31}$$

This is easily solved to find the general solution

$$y = Ae^{1000t} + Be^{-1000t} \quad . \tag{7.32}$$

Consider what will happen if you give this equation to an unsuspecting differential equation solver with the initial conditions on y and $\dot{y}$ chosen so that $A = 0$. It will start stepping along, making its usually harmless small errors at each step. However, a small error in y (and therefore $\dot{y}$) means that a tiny bit of the other solution will creep in, and

as soon as $A \neq 0$ then $\exp(1000t)$ will annihilate $\exp(-1000t)$ and the solution will blow up. The first thing to check when you run into a stiff differential equation is whether the variables can be rescaled so that their orders are comparable. Beyond that there is a range of special techniques for stiff differential equations that contain the different solutions; see [Gear, 1971].

This chapter has exclusively considered initial value problems. Sometimes *boundary-value* problems arise, in which values are known in the middle or the end of the interval. A classic example that was an important application for early computers was gunnery problems that seek initial conditions to launch a shell to land on a target. This is a harder task, and there are no simple solutions. One class of techniques, fittingly called *shooting methods*, sends multiple solutions across the interval and then tries to iteratively update its guess for the initial conditions that satisfy the boundary conditions. The other common approach is to use *finite elements* to discretize the entire interval to be solved and calculate it in parallel (Chapter 9).

7.4 SELECTED REFERENCES

[Press *et al*., 2007] Press, William H., Teukolsky, Saul A., Vetterling, William T., & Flannery, Brian P. (2007). *Numerical Recipes in C: The Art of Scientific Computing*. 3rd edn. New York, NY: Cambridge University Press.

As in so many other areas, the best first place to turn for numerical methods.

[Gear, 1971] Gear, C. William (1971). *Numerical Initial Value Problems in Ordinary Differential Equations*. Englewood Cliffs, NJ: Prentice-Hall.

[Stoer & Bulirsch, 2010] Stoer, J., & Bulirsch, R. (20). *Introduction to Numerical Analysis*. 3rd edn. New York, NY: Springer-Verlag. Translated by R. Bartels, W. Gautschi, and C. Witzgall.

These two are classic texts for differential equations.

[Young & Gregory, 1988] Young, David M., & Gregory, Robert Todd (1988). *A Survey of Numerical Mathematics*. New York, NY: Dover Publications. 2 volumes.

This is almost as broad in scope as [Press *et al*., 2007], but has more mathematical analysis of the algorithms in return for less practical guidance.

7.5 PROBLEMS

(7.1) What is the second-order approximation error of the *Heun* method, which averages the slope at the beginning and the end of the interval?

$$y(x+h) = y(x) + \frac{h}{2}\{f(x,y(x)) + f[x+h, y(x)+hf(x,y(x))]\} \tag{7.33}$$

(7.2) For a simple harmonic oscillator $\ddot{y}+y = 0$, with initial conditions $y(0) = 1,\ \dot{y}(0) = 0$, find $y(t)$ from $t = 0$ to 100π. Use an Euler method and a fixed-step fourth-order Runge–Kutta method. For each method check how the average local error, and the error in the final value and slope, depend on the step size.

(7.3) Write a fourth-order Runge–Kutta adaptive stepper for the preceding problem, and check how the average step size that it finds depends on the desired local error.

(7.4) Numerically solve the differential equation found in Problem 5.3:

$$l\ddot{\theta} + (g + \ddot{z}) \sin\theta = 0 \quad . \tag{7.34}$$

Take the motion of the platform to be periodic, and interactively explore the dynamics of the pendulum as a function of the amplitude and frequency of the excitation.