Solving ODE's with Matlab

Book · January 2003

DOI: 10.1017/CBO9780511615542 · Source: DBLP

3 authors, including:

Lawrence Shampine
Southern Methodist University
252 PUBLICATIONS **11,097** CITATIONS

SEE PROFILE

Skip Thompson
Radford University
43 PUBLICATIONS **1,138** CITATIONS

SEE PROFILE

Solving ODEs with MATLAB

L.F. Shampine and I. Gladwell
Department of Mathematics
Southern Methodist University
Dallas, TX 75275

S. Thompson
Department of Mathematics & Statistics
Radford University
Radford, VA 24142

Chapter 1

Initial Value Problems

1.1 Introduction

In this chapter we study the solution of initial value problems (IVPs) for ordinary differential equations (ODEs). Because ODEs arise in diverse forms, it is convenient for both theory and practice to write them in a standard form. It was shown in Chapter **??** how to prepare ODEs as a system of first order equations that in (column) vector notation has the form

$$y' = f(t, y) \tag{1.1}$$

With one exception, it is assumed throughout this chapter that the ODEs have this form. Because the MATLAB IVP solvers accept problems of the form $M(t, y)y' = f(t, y)$, it is discussed briefly in §1.3.2. In either case it is assumed that the ODEs are defined on a finite interval $a \leq t \leq b$ and that the initial values are provided as a vector

$$y(a) = A \tag{1.2}$$

The popular numerical methods for IVPs start with $y_0 = A = y(a)$ and compute successively approximations $y_n \approx y(t_n)$ on a mesh $a = t_0 < t_1 < \cdots < t_N = b$. On reaching t_n, the basic methods are distinguished by whether or not they use previously computed quantities such as $y_{n-1}, y_{n-2}, \ldots$. If they do, they are called *methods with memory* and otherwise, *one–step methods*. IVPs are categorized as non–stiff and stiff. It is hard to define stiffness, but its symptoms are easy to recognize. Unfortunately, the distinction between stiff and non–stiff IVPs can be very important when choosing a method. The MATLAB IVP solvers implement a variety of methods, but the documentation recommends that you try first `ode45`, a code based on a pair of one–step *explicit Runge–Kutta formulas*. If you suspect that the problem is stiff or if `ode45` should prove unsatisfactory, it is recommended that you try `ode15s`, a code based on the *backward differentiation formulas* (BDFs). These two types of methods are among the most widely used in general scientific computing, so we focus on them in our discussion of numerical methods. We do take up other methods to provide some perspective. In particular, we discuss the *Adams methods* that are implemented in `ode113`. They are often preferred over explicit Runge–Kutta methods when solving non–stiff problems in general scientific computing. The last part of this chapter is a tutorial that shows how to solve IVPs with the programs of MATLAB. You can read it and solve interesting problems in parallel with your reading about the theory of the various methods implemented in the programs.

1.2 Numerical Methods for IVPs

We focus on two kinds of methods for solving IVPs, the ones used by `ode45` and `ode15s`. They are explicit Runge–Kutta formulas for non–stiff IVPs and backward differentiation formulas for stiff IVPs, respectively. These methods are unquestionably among the most effective and widely used.

Comments are made about other methods where this helps put the developments in perspective. Although our discussion of the methods is brief, it does identify the most important issues for solving IVPs in practice.

Numerical solution of the IVP (1.1), (1.2) on the interval $a \le t \le b$ proceeds in steps. Starting with the initial value $y_0 = A$, values $y_n \approx y(t_n)$ are computed successively on a mesh

$$a = t_0 < t_1 < \cdots < t_N = b$$

The computation of y_{n+1} is often described as taking a step of size $h_n = t_{n+1} - t_n$ from t_n. For brevity we generally write $h = h_n$ in discussing the step from t_n. On reaching (t_n, y_n), the *local solution* $u(t)$ is defined as the solution of

$$u' = f(t, u), \qquad u(t_n) = y_n$$

A standard result from the theory of ODEs states that if $v(t)$ and $w(t)$ are solutions of (1.1) and if $f(t, y)$ satisfies a Lipschitz condition with constant L, then for $\alpha < \beta$,

$$\|v(\beta) - w(\beta)\| \le \|v(\alpha) - w(\alpha)\| e^{L(\beta-\alpha)}$$

In the *classical situation* that $L(b-a)$ is of modest size, this result tells us that the IVP (1.1), (1.2) is moderately stable. This is only a sufficient condition. Indeed, *stiff* problems are (very) stable, yet $L(b-a) \gg 1$. Without doing some computation, it is not easy to recognize that a stable IVP is stiff. There are two essential properties that will help you with this: A stiff problem is very stable in the sense that some solutions of the ODE starting near the solution of interest converge to it very rapidly. "Very rapidly" here means that the solutions converge over a distance that is small compared to $b-a$, the length of the interval of integration. This property implies that some solutions change very rapidly, but the second property is that the solution of interest is slowly varying.

The basic numerical methods approximate the solution only on a mesh, but in some codes, including all of the MATLAB solvers, they are supplemented with (inexpensive) methods for approximating the solution between mesh points. The BDFs are based on polynomial interpolation and so give rise immediately to a continuous piecewise–polynomial function $S(t)$ that approximates $y(t)$ everywhere in $[a, b]$. There is no natural polynomial interpolant for explicit Runge–Kutta methods, which is why such interpolants are a relatively new development. A method that approximates $y(t)$ on each step $[t_n, t_{n+1}]$ by a polynomial that interpolates the approximate solution at the end points of the interval is called a *continuous extension* of the Runge–Kutta formula. We'll use the term more generally to refer to a piecewise–polynomial approximate solution $S(t)$ defined in this way on all of $[a, b]$.

1.2.1 One–Step Methods

One–step methods use only data gathered in the current step. MATLAB includes solvers based on a number of different kinds of one–step methods, but we concentrate on one kind, explicit Runge–Kutta methods. In the course of our study of these explicit methods, we'll also develop some implicit methods. They are widely used for solving BVPs, so we return to them in Chapter **??**.

It is illuminating first to take up the special case of *quadrature*

$$y' = f(t), \qquad y(a) = A \tag{1.3}$$

Certainly we must be able to deal with these simpler problems and because they are simpler, it is easier to explain the methods. The local solution at t_n satisfies

$$u' = f(t), \qquad u(t_n) = y_n$$

so

$$u(t_n + h) = y_n + \int_{t_n}^{t_n+h} f(x)\, dx$$

Computing $y_{n+1} \approx u(t_n + h)$ amounts to approximating numerically a definite integral.

A basic tactic in numerical analysis is this: If you cannot do what you want with a function $f(x)$, approximate it with an interpolating polynomial $P(x)$ and use the polynomial instead. The popular formulas for approximating definite integrals can be derived in this way. For instance, if we approximate the function $f(x)$ on the interval $[t_n, t_n + h]$ by interpolating it with the constant polynomial $P(x) = f(t_n)$, we can integrate the polynomial to obtain

$$\int_{t_n}^{t_n+h} f(x)\,dx \approx \int_{t_n}^{t_n+h} P(x)\,dx = hf(t_n)$$

Or, if we interpolate at the other end of the interval,

$$\int_{t_n}^{t_n+h} f(x)\,dx \approx \int_{t_n}^{t_n+h} P(x)\,dx = hf(t_{n+1})$$

Similarly, if we use a linear polynomial that interpolates $f(x)$ at both ends of the interval,

$$P(x) = \left(\frac{(t_n + h) - x}{h}\right) f(t_n) + \left(\frac{x - t_n}{h}\right) f(t_n + h)$$

integrating we obtain the approximation

$$\int_{t_n}^{t_n+h} f(x)\,dx \approx \frac{h}{2}\left[f(t_n) + f(t_n + h)\right]$$

Geometrically the first two schemes approximate the integral by the area of a rectangle. The third is known as the trapezoidal rule because it approximates the integral by the area of a trapezoid.

We can deduce the accuracy of these approximations for a smooth function $f(x)$ by a standard result from polynomial interpolation theory [113]: If $P(x)$ is the unique polynomial of degree less than s that interpolates a smooth function $f(x)$ at s distinct nodes $t_{n,j} = t_n + \alpha_j h$ in the interval $[t_n, t_n + h]$,

$$P(t_{n,j}) = f(t_{n,j}), \quad j = 1, 2, \ldots, s$$

then

$$P(x) = \sum_{j=1}^{s} f_{n,j} \prod_{i=1, i\neq j}^{s} \frac{x - t_{n,i}}{t_{n,j} - t_{n,i}}$$

and for each point x in the interval $[t_n, t_n + h]$, there is a point ξ in the same interval for which

$$f(x) - P(x) = \frac{f^{(s+1)}(\xi)}{(s+1)!} \prod_{j=1}^{s} (x - t_{n,j})$$

For a function $f(x)$ that is sufficiently smooth, the derivative appearing in this expression is bounded in magnitude on the interval. It is then easy to see that there is a constant C such that

$$|f(x) - P(x)| \leq Ch^s$$

for all $x \in [t_n, t_n + h]$. A standard notation and terminology is used when we focus our attention on the behavior with respect to the step size h, ignoring the value of the constant. We write

$$f(x) - P(x) = O(h^s)$$

or equivalently, $f(x) = P(x) + O(h^s)$. We say that the difference between $f(x)$ and $P(x)$ is "big oh of h to the s" or for short, the difference is of *order s*. With the theorem about the accuracy of polynomial interpolation, it is easy to see that

$$\int_{t_n}^{t_n+h} f(x)\,dx = \int_{t_n}^{t_n+h} P(x)\,dx + O(h^{s+1})$$

Applying these results to the rectangle approximations to the integral gives the formula

$$y_{n+1} = y_n + hf(t_n) \tag{1.4}$$

for which the local error

$$u(t_n + h) - y_{n+1} = \int_{t_n}^{t_n+h} f(x)\,dx - hf(t_n) = O(h^2)$$

and

$$y_{n+1} = y_n + hf(t_{n+1}) \tag{1.5}$$

which has the same order of accuracy. The trapezoidal rule

$$y_{n+1} = y_n + h\left[\frac{1}{2}f(t_n) + \frac{1}{2}f(t_n + h)\right] \tag{1.6}$$

has local error $u(t_n + h) - y_{n+1} = O(h^3)$. More generally, polynomial interpolation at s nodes leads to an *interpolatory quadrature formula* of the form

$$\int_{t_n}^{t_n+h} f(x)\,dx = h\sum_{j=1}^{s} A_j f(t_n + \alpha_j h) + O(h^{p+1})$$

and to a numerical method

$$y_{n+1} = y_n + h\sum_{j=1}^{s} A_j f(t_n + \alpha_j h) \tag{1.7}$$

for which the local error is $O(h^{p+1})$. Interpolation theory assures us that the order $p \geq s$, but an important fact in the practical approximation of integrals is that for some choices of nodes, the order $p > s$. For instance, the midpoint rule that comes from a constant polynomial that interpolates $f(x)$ at $t_n + 0.5h$ has order $p = 2$ instead of the value $p = 1$ that we might expect on general grounds.

We have obtained specific formulas of the form (1.7) by integrating an interpolating polynomial. Let us now start with a formula of this general form and ask how we might choose the coefficients A_j to find an accurate formula. To determine how accurate the formula is, we'll expand both $u(t_n + h)$ and y_{n+1} in Taylor series about t_n and see how many terms agree. Using the IVP satisfied by the local solution, we find that

$$\begin{aligned} u(t_n + h) &= u(t_n) + \sum_{k=1}^{p} h^k \frac{u^{(k)}(t_n)}{k!} + O(h^{p+1}) \\ &= y_n + \sum_{k=1}^{p} h^k \frac{f^{(k-1)}(t_n)}{k!} + O(h^{p+1}) \end{aligned}$$

Expanding the formula on the right hand side of (1.7) is a little more complicated. First using Taylor series we expand

$$f(t_n + \alpha_j h) = \sum_{r=0}^{p-1} (\alpha_j h)^r \frac{f^{(r)}(t_n)}{r!} + O(h^p)$$

and then we substitute this result into the formula to obtain

$$\begin{aligned} y_{n+1} &= y_n + h\sum_{j=1}^{s} A_j \left(\sum_{k=1}^{p} (\alpha_j h)^{k-1} \frac{f^{(k-1)}(t_n)}{(k-1)!} \right) + O(h^{p+1}) \\ &= y_n + \sum_{k=1}^{p} h^k \left(\sum_{j=1}^{s} A_j \alpha_j^{k-1} \right) \frac{f^{(k-1)}(t_n)}{(k-1)!} + O(h^{p+1}) \end{aligned}$$

Comparing the two expansions, we see that $u(t_n + h) = y_{n+1} + O(h^{p+1})$ if, and only if,

$$\frac{1}{k} = \sum_{j=1}^{s} A_j \alpha_j^{k-1}, \quad k = 1, 2, \ldots, p \tag{1.8}$$

If we use only one node, that is $s = 1$, the first equation of (1.8) requires that $A_1 = 1$. With this value for A_1, the second equation is

$$\frac{1}{2} = A_1 \alpha_1 = \alpha_1$$

If $\alpha_1 \neq \frac{1}{2}$, this equation is not satisfied and $u(t_n + h) = y_{n+1} + O(h^2)$. If $\alpha_1 = \frac{1}{2}$, the equation is satisfied and the third equation becomes

$$\frac{1}{3} = 1 \times \left(\frac{1}{2}\right)^2$$

This equation is not satisfied, so $u(t_n + h) = y_{n+1} + O(h^3)$ for this formula. The formula is the midpoint rule that we just met. Exercise 1.1 asks you to verify the order of two other formulas that are based on important quadrature rules.

We begin a discussion of the convergence of these formulas by working out the stability of the ODE. The function f satisfies a Lipschitz condition with $L = 0$, so the general result stated earlier tells us that if $v(t)$ and $w(t)$ are solutions of $y' = f(t)$ and $\alpha < \beta$, then

$$\|v(\beta) - w(\beta)\| \leq \|v(\alpha) - w(\alpha)\|$$

However, it is perfectly easy to prove a stronger result directly. A solution $v(t)$ of $y' = f(t)$ has the form

$$v(t) = v(\alpha) + \int_{\alpha}^{t} f(x)\, dx$$

Evaluating this expression at $t = \beta$ and subtracting a similar expression for $w(t)$ leads to

$$v(\beta) - w(\beta) = v(\alpha) - w(\alpha)$$

From this we see that the local error of a step from t_n moves us to a solution of the ODE that is parallel to the solution through y_n.

Let the true error at t_n be

$$e_n = y(t_n) - y_n$$

By choosing $y_0 = A$, we have $e_0 = 0$. In Chapter **??** we studied the propagation of error by writing

$$e_{n+1} = y(t_{n+1}) - y_{n+1} = [u(t_{n+1}) - y_{n+1}] + [y(t_{n+1}) - u(t_{n+1})]$$

The first term on the right is the local error that we assume is bounded in magnitude by Ch_n^{p+1}. Using the general bound on stability for ODEs that satisfy Lipschitz conditions, the second term is bounded by

$$|y(t_{n+1}) - u(t_{n+1})| \leq |y(t_n) - y_n|\, e^{Lh_n}$$

Here we use the definition $u(t_n) = y_n$. Putting these bounds together, we obtain

$$|e_{n+1}| \leq Ch_n^{p+1} + |e_n|\, e^{Lh_n}$$

The error in this step comes from two sources. One is the local error introduced at each step by the numerical method. The other is amplification of the error from preceding steps due to the stability of the IVP itself. The net effect is particularly easy to understand for quadrature problems because there is no amplification and the local errors just add up in this bound. If we solve a quadrature problem with a constant step size $h = \frac{b-a}{N}$, we have a uniform bound

$$|y(t_n) - y_n| = |e_n| \leq nCh^{p+1} \leq (b - a)Ch^p$$

That is, $y_n = y(t_n) + O(h^p)$ for all n. For this reason, when the local error is $O(h^{p+1})$, we say that the formula is of order p because that is the order of approximation to $y(t)$. When the step size varies, it is easy to modify this proof to see that if H is the maximum step size, the true (global) error is $O(H^p)$.

Local Error Estimation

The local error of the result y_{n+1} of a formula of order p is

$$le_n = u(t_n + h) - y_{n+1}$$

If we also apply a formula of order $p+1$ to compute a result y^*_{n+1} on this step, we can form

$$\begin{aligned} est &= y^*_{n+1} - y_{n+1} \\ &= [u(t_n + h) - y_{n+1}] - [u(t_n + h) - y^*_{n+1}] \\ &= le_n + O(h^{p+2}) \end{aligned} \tag{1.9}$$

This is a computable estimate of the local error of the lower order formula because le_n is $O(h^{p+1})$ and so dominates in (1.9) for small enough values of h. Put differently, we can estimate the error in y_{n+1} by comparing it to the more accurate approximate solution y^*_{n+1}. Generally the most expensive part of taking a step is forming the function values $f(t_n + \alpha_j h)$, so the trick to making local error estimation practical is to find a pair of formulas that share as many of these function evaluations as possible. For the estimate to be any good, the higher order result y^*_{n+1} has to be the more accurate. But if that is so, why would we discard it in favor of using y_{n+1} to advance the integration? Advancing the integration with the more accurate result y^*_{n+1} is called *local extrapolation*. Most of the popular explicit Runge–Kutta codes use local extrapolation and in particular, the MATLAB solvers `ode23` and `ode45` use it. In this way of proceeding, we do not know precisely how small the local error is at each step, but we believe that it is rather smaller than the estimated local error.

Solvers for IVPs control the estimated local error. A local error tolerance τ is specified and if the estimated error is too large relative to this tolerance, the step is rejected and another attempt is made with a smaller step size. In our expansion of the local error, we just worked out the order of the first non–zero term. If we carry another term in the expansion, we find that

$$u(t_n + h) - y_{n+1} = h^{p+1}\phi(t_n) + O(h^{p+2}) \tag{1.10}$$

Using this, we can see how to adjust the step size. If we were to try again to take a step from t_n with step size σh, the local error would be

$$\begin{aligned} (\sigma h)^{p+1}\phi(t_n) + O((\sigma h)^{p+2}) &= \sigma^{p+1}h^{p+1}\phi(t_n) + O(h^{p+2}) \\ &= \sigma^{p+1} est + O(h^{p+2}) \end{aligned} \tag{1.11}$$

The largest step size that we predict will pass the error test corresponds to choosing σ so that $|\sigma^{p+1} est| \approx \tau$. This step size is

$$h\left(\frac{\tau}{|est|}\right)^{1/(p+1)}$$

The solver is required to find a step size for which the magnitude of the estimated local error is no larger than the tolerance, so it must keep trying until it succeeds or it gives up. It might give up because it has done too much work. It also might give up because it finds that it needs a step size too small for the precision of the computer. This is much like asking for an impossible relative accuracy, an issue discussed in Chapter **??**. On the other hand, if the step is a success and the local error is rather smaller than necessary, we might increase the step size for the *next* step. This makes the computation more efficient because larger step sizes mean that we reach the end of the interval of integration in fewer steps. The same recipe can be used to estimate the step size that might be used on the next step: We predict that the error of a step of size σh taken from t_{n+1} would be

$$\begin{aligned} u(t_{n+1} + \sigma h) - y_{n+2} &= (\sigma h)^{p+1}\phi(t_{n+1}) + O(h^{p+2}) \\ &= \sigma^{p+1}h^{p+1}\phi(t_n) + O(h^{p+2}) \\ &= \sigma^{p+1} est + O(h^{p+2}) \end{aligned} \tag{1.12}$$

In general terms, this is the way that popular codes select the step size, but we have omitted important practical details. For instance, how much the step size is increased or decreased must be

limited because we cannot neglect the effects of higher order terms (the "big oh" terms in equations (1.11) and (1.12)) when the change of step size is large. Also, several approximations are made in predicting the step size, so the estimate should not be taken too seriously. Because a failed step is relatively expensive and because the error estimate used to predict the step size may not be very reliable, the codes use a fraction of the predicted step size. Fractions like 0.8 and 0.9 are commonly used. The aim is to achieve the required accuracy without too great a chance of a failed step.

In the early days of numerical computing when it was not understood how to estimate and control the local error, a constant step size was used. This approach is still seen today, but estimation and control of the local error is extremely important in practice, so important that it is used in all the computations of this book except when we wish to make a specific point about constant step size integration. Estimation and control of the local error is what gives us some confidence that we have computed a meaningful approximation to the solution of the IVP. Moreover, estimating the local error is not expensive. Generally it more than pays for itself because the step size is then not restricted to the smallest necessary to resolve the behavior of the solution over the whole interval of integration or to ensure stability of the integration. Indeed, it is impractical to solve many IVPs, including all stiff problems, with constant step size. Recall the proton transfer problem of §**??**. Its solution has a boundary layer of width about 10^{-10}. Clearly we must use some steps of this general size to resolve the boundary layer, but the problem is posed on an interval of length about 10^6. If we were to use a constant step size for the whole integration, we would need something like 10^{16} steps to solve this problem! This would take an impractically long time even on today's fastest computers, and even if it were possible to perform the calculation, the numerical solution would be dominated by the cumulative effects of roundoff error. In our solution of this problem for Figs. **??** and **??**, the integration required only 10^2 steps. The step sizes ranged in size from $7 \cdot 10^{-14}$ in the boundary layer to $4 \cdot 10^4$ where the solution is slowly varying. The IVP was solved in a few seconds on a PC Pentium II 433Mhz and there were no obvious effects of roundoff error.

Runge–Kutta Methods

We have now a brief, yet nearly complete description of solving quadrature problems in the manner of a modern explicit Runge–Kutta code for IVPs. General IVPs are handled in much the same way, so mostly we point out differences. Now the local solution satisfies the IVP

$$u' = f(t, u), \qquad u(t_n) = y_n$$

Again we integrate to obtain

$$u(t_n + h) = y_n + \int_{t_n}^{t_n+h} f(x, u(x))\, dx$$

The crucial difference is that now the unknown local solution appears on both sides of the equation. If we approximate the integral with a quadrature formula, we have

$$\int_{t_n}^{t_n+h} f(x, u(x))\, dx = h \sum_{j=1}^{s} A_j f(t_{n,j}, u(t_{n,j})) + O(h^{p+1}) \tag{1.13}$$

where again we abbreviate $t_n + \alpha_j h = t_{n,j}$. This doesn't seem like much help because we don't know the intermediate values $u(t_{n,j})$. Before discussing the two basic approaches to dealing with this problem, we consider some important examples of formulas for which there is no intermediate value.

The first (left) rectangle approximation to the integral is

$$\int_{t_n}^{t_n+h} f(x, u(x))\, dx = hf(t_n, u(t_n)) + O(h^2) = hf(t_n, y_n) + O(h^2)$$

The corresponding formula

$$y_{n+1} = y_n + hf(t_n, y_n) \tag{1.14}$$

is known as the *(forward) Euler method*. In the context of solving time–dependent partial differential equations (PDEs) it is known as the *fully explicit method*. The approximation

$$y_{n+1} = u(t_n + h) + O(h^2)$$

so it is a first order explicit Runge–Kutta formula. The second (right) rectangle approximation leads to the *backward Euler method*

$$y_{n+1} = y_n + hf(t_{n+1}, y_{n+1}) \tag{1.15}$$

which is known for PDEs as the *fully implicit method*. Here we see a major difficulty that is not present for quadrature problems—the new approximate solution y_{n+1} is defined implicitly as the solution of a set of algebraic equations. This formula is no more accurate than the forward Euler method, so why would we bother with the expense of evaluating an implicit formula? One answer is to overcome stiffness. As it happens, the backward Euler method is the lowest order member of the family of backward differentiation formulas (BDFs) that we derive from a different point of view in the next section. The member of this family that is of order k is denoted by BDFk, so the backward Euler method is also known as BDF1.

The trapezoidal rule

$$y_{n+1} = y_n + h\left[\frac{1}{2}f(t_n, y_n) + \frac{1}{2}f(t_{n+1}, y_{n+1})\right] \tag{1.16}$$

is a second order implicit Runge–Kutta formula implemented in the MATLAB IVP solver `ode23t`. In the context of PDEs it is called the *Crank–Nicolson method*. Notice that the trapezoidal rule treats the solution values y_n and y_{n+1} in the same way. A formula like this is said to be symmetric. There is a direction of integration when solving IVPs, but usually not when solving BVPs. Because symmetric formulas do not have a preferred direction, they are widely used to solve BVPs.

Returning to the issue of intermediate values, suppose that we already have a formula that we can use to compute approximations $y_{n,j} = u(t_{n,j}) + O(h^p)$. Stating this assumption about the accuracy more formally, we assume there is a constant C such that

$$\|u(t_{n,j}) - y_{n,j}\| \le Ch^p$$

Along with our assumption that the ODE function f satisfies a Lipschitz condition with constant L, this implies that

$$\|f(t_{n,j}, u(t_{n,j})) - f(t_{n,j}, y_{n,j})\| \le L\|u(t_{n,j}) - y_{n,j}\| \le LCh^p \tag{1.17}$$

If we replace the function evaluations $f(t_{n,j}, u(t_{n,j}))$ with the computable approximations $f(t_{n,j}, y_{n,j})$ a little manipulation of (1.13) and (1.17) shows that

$$\int_{t_n}^{t_n+h} f(x, u(x))\,dx = h\sum_{j=1}^{s} A_j f(t_{n,j}, y_{n,j}) + O(h^{p+1})$$

In this way we arrive at a formula

$$y_{n+1} = y_n + h\sum_{j=1}^{s} A_j f(t_{n,j}, y_{n,j})$$

We see that if we already know formulas that we can use to compute intermediate values $y_{n,j}$ accurate to $O(h^p)$, we have constructed a formula to compute an approximate solution y_{n+1} accurate to $O(h^{p+1})$.

There are two basic approaches to choosing the formulas for computing intermediate values. One is to use formulas of the same form as that for computing y_{n+1}. This leads to an implicit Runge–Kutta formula, a system of algebraic equations that generally involves computing simultaneously the approximate solution y_{n+1} and the intermediate values $y_{n,j}$ for $j = 1, 2, \ldots, s$. To be useful for the solution of stiff IVPs, a formula must be implicit to some degree, but it is not necessary that all the $y_{n,j}$ be computed simultaneously. The MATLAB IVP solver `ode23tb` is based on an implicit formula that computes the intermediate values one at a time. If we choose explicit formulas for all the $y_{n,j}$, we obtain an explicit formula for y_{n+1}. Explicit formulas are popular for the solution of non–stiff problems. The MATLAB IVP solvers `ode23` and `ode45` are based on formulas of this kind.

Explicit Runge–Kutta Formulas

Using the explicit forward Euler method, we can form the intermediate values needed for any quadrature formula with $p = 2$ to obtain a formula for which $u(t_n + h) = y_{n+1} + O(h^3)$. For example, if the quadrature formula is the trapezoidal rule, we obtain a second order method called *Heun's method*

$$\begin{aligned} y_{n,1} &= y_n + hf(t_n, y_n) \\ y_{n+1} &= y_n + h\left[\frac{1}{2}f(t_n, y_n) + \frac{1}{2}f(t_{n+1}, y_{n,1})\right] \end{aligned} \tag{1.18}$$

Exercise 1.10 asks you to solve an IVP with this formula. In Exercise 1.2 you are asked to construct a formula of the same order of accuracy using the midpoint rule instead of the trapezoidal rule. Using Heun's method, we can produce the intermediate values needed for any quadrature formula with order $p = 3$ to obtain a formula of order 4, and so forth. Because we can construct interpolatory quadrature formulas of any order, we see now how to construct an explicit Runge–Kutta formula of any order by this "bootstrapping" technique.

When we take account of the intermediate values, we find that the formulas resulting from the construction just outlined have the form of an explicit recipe that starts with

$$y_{n,1} = y_n, \qquad f_{n,1} = f(t_n, y_{n,1}) \tag{1.19}$$

then, for $j = 2, 3, \ldots, s$ forms

$$y_{n,j} = y_n + h_n \sum_{k=1}^{j-1} \beta_{j,k} f_{n,k}, \qquad f_{n,j} = f(t_n + \alpha_j h_n, y_{n,j}) \tag{1.20}$$

and finishes with

$$y_{n+1} = y_n + h_n \sum_{j=1}^{s} \gamma_j f_{n,j} \tag{1.21}$$

The values (1.19) are the degenerate case $j = 1$ in (1.20) with $\alpha_1 = 0$, but they are written separately here to remind us that the method starts with the value $u(t_n) = y_n$ and slope

$$u'(t_n) = f(t_n, u(t_n)) = f(t_n, y_n)$$

of the local solution at the beginning of the step. A useful measure of the work involved in evaluating such an explicit formula is the number of evaluations of the function $f(x, y)$, that is, the number of *stages* $f_{n,j}$. Here the number of stages is s.

In principle we can work out the order of the formula given by equations (1.19), (1.20), and (1.21) just as we did with the formula (1.7) for quadrature problems. Indeed, if we apply the formula to a quadrature problem, the conditions (1.8) are in the present notation

$$\frac{1}{k} = \sum_{j=1}^{s} \gamma_j \alpha_j^{k-1}, \quad k = 1, 2, \ldots, p \tag{1.22}$$

The conditions on the coefficients of a Runge–Kutta formula for it to be of order p are called the *equations of condition*. The equations (1.22) are a subset, so they are necessary, but they are certainly not sufficient. Two matters complicate the argument in the general case. One is the presence of the solution $u(t)$ in the derivative $f(t, u(t))$ and the other is that $u(t)$ is a vector. To understand this better, let's look at the first non–trivial term in expanding $u(t_n + h)$ about t_n. In the Taylor series

$$u(t_n + h) = u(t_n) + u'(t_n)h + u''(t_n)\frac{h^2}{2} + \ldots$$

the IVP provides us immediately with $u(t_n) = y_n$ and $u'(t_n) = f(t_n, u(t_n)) = f(t_n, y_n)$. The next term is more complicated. If there are d equations, component i of the local solution satisfies

$$u_i'(t) = f_i(t, u_1(t), u_2(t), \ldots, u_d(t))$$

It is then straightforward to obtain

$$u_i''(t) = \frac{\partial f_i}{\partial t} + \sum_{j=1}^{d} \frac{\partial f_i}{\partial u_j}\frac{du_j}{dt} = \frac{\partial f_i}{\partial t} + \sum_{j=1}^{d} \frac{\partial f_i}{\partial u_j} f_j$$

Clearly to deal with formulas of even moderate orders, we need to develop ways of making the manipulations easier. It would simplify the expressions considerably if the function $f(t, y)$ did not depend on t. Such a problem is said to be in *autonomous* form. Because it is always possible to obtain an equivalent IVP in autonomous form, the theory often assumes it, though most codes do not. We digress for a moment to show the usual way this is done for the IVP

$$\frac{dy}{dt} = f(t, y), \qquad y(a) = A$$

on the interval $a \le t \le b$. If we change to the new independent variable $x = t$ and make t a dependent variable,

$$\frac{dY}{dx} = \frac{d}{dx}\begin{pmatrix} y \\ t \end{pmatrix} = \begin{pmatrix} f(t,y) \\ 1 \end{pmatrix} = F(Y), \qquad Y(a) = \begin{pmatrix} A \\ a \end{pmatrix}$$

on $a \le x \le b$. This approach is convenient because all standard methods integrate the equation for t exactly. Returning now to the derivation of methods, it is clear that a more powerful notation is needed, as well as recursions for computing derivatives. This is all rather technical and we have no need for the details, so we take the special case of quadrature as representative. To gain an appreciation of the general case, Exercise 1.3 asks you to work out the details for a second order explicit Runge–Kutta formula.

We have seen how to construct explicit Runge–Kutta formulas of any order, but generally the formulas constructed this way are not very efficient. Much effort has been devoted to finding formulas that yield a given order p with as few stages as possible. For orders $p = 1$, 2, 3, and 4, the minimum number of stages is $s = p$ and for order $p = 5$, it is $s = 6$. The minimum number of stages is interesting, but not as important as it might seem. Extra stages can be used to find a more accurate formula. With a more accurate formula, you can take larger steps, enough larger that you might be able to solve a problem with fewer overall evaluations of $f(t, y)$ even though each step is more expensive. Besides this basic point, the issue is not the cost of evaluating a formula by itself, rather the cost of evaluating a pair of formulas for taking a step and estimating the local error. The argument we made in deriving the estimate (1.9) of the local error was not restricted to quadrature problems. Deriving pairs of formulas that share many of their stages is a challenging task. We have remarked that at least $s = 6$ stages are required for a formula of order $p = 5$. Pairs of formulas of orders 4 and 5, denoted a (4,5) pair, are known that require a total of 6 stages. For example, a pair denoted F(4,5) due to Fehlberg [41] using a total of 6 stages is in wide use. The popular (4,5) pair due to Dormand and Prince [35], called DOPRI5, that is implemented in `ode45` has 7 stages. The additional stage is used to improve the formula and in tests DOPRI5 has proved somewhat superior to F(4,5).

The coefficients defining a Runge–Kutta formula are commonly presented as in Table 1.1, a notation due to Butcher. For an explicit Runge–Kutta method all entries on and above the diagonal of the matrix β are zero and it is conventional not to display them. When presenting pairs of formulas, the vectors of coefficients γ are presented one above the other in the tableau. We have already seen a simple example of a pair with a minimal number of stages, namely the (1,2) pair consisting of the Euler and Heun formulas. Table 1.2 displays the tableau for this pair.

α	β
	γ

Table 1.1: Butcher tableau for Runge–Kutta formulas.

0			
1	1		
	1		
	$\frac{1}{2}$	$\frac{1}{2}$	

Table 1.2: The Euler–Heun (1,2) pair.

The Bogacki–Shampine [20] BS(2,3) pair implemented in `ode23` is displayed in Table 1.3. The layout is a little different because the BS(2,3) pair exemplifies a technique called *First Same as Last* (FSAL). For an FSAL formula, the First stage of the next step is the Same As the Last stage of the current step. The last line of the table displays the coefficients γ for the second order formula of 4 stages. The line just above it contains the coefficients for the last stage, which by construction is a third order formula of 3 stages. The way this works is that you form a third order result y_{n+1} with three stages, evaluate the fourth stage $f(t_{n+1}, y_{n+1})$, and then form the second order result. This pair was designed to be used with local extrapolation, which is to say that the integration is to be advanced with the third order result, y_{n+1}, as the approximate solution. The stage formed for the evaluation of the second order formula and the error estimate is the first stage of the next step. In this way we get a stage for "free" if the step is accepted. In practice, most steps are accepted so this pair costs little more than a pair involving just three stages. Clearly, deriving pairs constrained to have the FSAL property is a challenge, but some popular pairs are of this form. One such is the 7 stage DOPRI5 pair mentioned earlier as the pair implemented in `ode45`. In practice it costs little more than the minimum of 6 stages per step that are required for any fifth order formula. The widely used Fortran 77 package `RKSUITE` [17] and its close relative the Fortran 90 package `rksuite_90` [16] implement a (4,5) FSAL pair that has still another stage. The topic of formula pairs is taken up in Exercise 1.4.

0				
$\frac{1}{2}$	$\frac{1}{2}$			
$\frac{3}{4}$	0	$\frac{3}{4}$		
1	$\frac{2}{9}$	$\frac{1}{3}$	$\frac{4}{9}$	
	$\frac{7}{24}$	$\frac{1}{4}$	$\frac{1}{3}$	$\frac{1}{8}$

Table 1.3: The BS(2,3) pair.

Continuous Extensions

We have been discussing the approximation of $y(t)$ at points $t_0 < t_1 < \cdots$, but for some purposes it is valuable to have an approximation for all t. For example, plot packages draw straight lines between data points. Runge–Kutta formulas of moderate to high order take such long steps that it is quite common for straight line segments to be noticeable and distracting in plots of solution components.

Figs. 4.28 and 4.29 of Bender and Orszag [14] provide good examples of this in the literature and Exercise 1.21 provides another example. To plot a smooth graph, we need an inexpensive way of approximating the solution between mesh points. This is often called *dense output* in the context of Runge–Kutta methods.

Continuous extensions are a relatively recent development in the theory of Runge–Kutta methods. The idea is that after taking a step from t_n to $t_n + h$, the stages used in computing the solution and the error estimate, and perhaps a few additional stages, are used to determine a polynomial approximation to $u(t)$ throughout $[t_n, t_n + h]$. The scheme used in `ode23` is particularly simple. At the beginning of the step we have approximations y_n and $y'_n = f(t_n, y_n)$ to the value and slope of the solution there. At the end of the step we compute an approximation y_{n+1}. The BS(2,3) pair also evaluates an approximation to the slope $y'_{n+1} = f(t_{n+1}, y_{n+1})$ because it is FSAL. (This information is readily available for any explicit Runge–Kutta formula because this slope is always the first stage of the next step.) With approximations to value and slope at both ends of an interval, we can use cubic Hermite interpolation to the solution and its slope at both ends of the current step. Using interpolation theory, it can be shown that this cubic interpolating polynomial approximates the local solution as accurately at all points of $[t_n, t_{n+1}]$ as y_{n+1} approximates it at t_{n+1}. The cubic Hermite interpolant on $[t_{n-1}, t_n]$ has the same value and slope at t_n as the interpolant on $[t_n, t_{n+1}]$, so this construction provides a piecewise-cubic polynomial $S(t) \in C^1[a, b]$. This approximation underlies the `dde23` code for solving delay differential equations that we study in Chapter **??**.

The interpolation approach to continuous extensions of Runge–Kutta formulas is valuable, but somewhat limited. For each $\sigma \in [0, 1]$, an interpolant evaluated at $t_n + \sigma h$ can be viewed as a Runge–Kutta formula for taking a step of size σh from t_n. Another way to proceed is to derive such a family of formulas directly. The trick is to find a formula for taking a step of size σh that shares as many stages as possible with the formula that we use to step to $t_n + h$. The new formula depends on σ, but these stages do not and if additional stages are needed to achieve the desired order, they should also not depend on σ. It is possible to derive such families of formulas with coefficients that are polynomials in σ. The continuous extension of `ode45` was derived in this way. No extra stages are needed to form its polynomial approximation of fourth order, but the piecewise–polynomial interpolant $S(t)$ is only continuous on the interval $[a, b]$. The DOPRI5 formula of `ode45` allows the solver to take such large steps that by default the solver evaluates the continuous extension at four equally spaced points in the span of every step and returns them along with the approximations computed directly by the formula. Generally these additional solution values are sufficient to provide a smooth graph. An option is provided for increasing the number of output points if necessary.

Exercise 1.1. Using the equations of condition (1.8) for quadrature problems, verify that the following methods are of order 4:

- Simpson's method is based on Simpson's quadrature formula, also known as the three point Lobatto formula,
$$\int_a^b f(x)\,dx = \frac{b-a}{6}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right]$$
- The two point Gaussian quadrature method is based on the formula
$$\int_a^b f(x)\,dx = \frac{b-a}{2}\left[f\left(\frac{a+b}{2} - \frac{b-a}{2\sqrt{3}}\right) + f\left(\frac{a+b}{2} + \frac{b-a}{2\sqrt{3}}\right)\right]$$

Exercise 1.2. Use Euler's method and the midpoint rule to derive a two stage, second order, explicit Runge–Kutta method.

Exercise 1.3. To understand better the equations of condition, derive the three equations for a formula of the form
$$y_{n+1} = y_n + h\left[\gamma_1 f_{n,1} + \gamma_2 f_{n,2}\right]$$
where
$$f_{n,1} = f\left(t_n, y_n\right)$$
$$f_{n,2} = f\left(t_n + \alpha_1 h, y_n + h\beta_{1,0} f_{n,1}\right)$$

to be of second order. In the step from (t_n, y_n), the result y_{n+1} of the formula is to approximate the solution of

$$u' = f(t,u), \qquad u(t_n) = y_n$$

at $t_{n+1} = t_n + h$. Expand $u(t_{n+1})$ and y_{n+1} about t_n in powers of h and equate terms to compute the equations of condition. To simplify the expansions, do this for a scalar function $f(t,u)$.

Exercise 1.4. The explicit Runge–Kutta formulas

$$\begin{aligned} y_{n+1} &= y_n + hf_{n,2} \\ y^*_{n+1} &= y_n + \frac{h}{9}\left[2f_{n,1} + 3f_{n,2} + 4f_{n,3}\right] \end{aligned}$$

of three stages

$$\begin{aligned} f_{n,1} &= f(t_n, y_n) \\ f_{n,2} &= f(t_n + \frac{h}{2}, y_n + \frac{h}{2}f_{n,1}) \\ f_{n,3} &= f(t_n + \frac{3}{4}h, y_n + \frac{3}{4}hf_{n,2}) \end{aligned}$$

are of order 2 and 3, respectively. State this (2,3) pair as a Butcher tableau. The local error of the lower order formula is estimated by $est = y^*_{n+1} - y_{n+1}$. Suppose that the integration is to be advanced with the higher order result (local extrapolation) and that you are given a relative error tolerance τ_r and absolute error tolerance τ_a. This means that you will accept the step if

$$|est| \le \tau_r \, |y^*_{n+1}| + \tau_a$$

What step size $h_{new} = \sigma h$ should you use if you must repeat the step because the estimated local error is too large? What step size should you use for the next step if the estimated local error is acceptable? Some solvers measure the error relative to $0.5(|y_n| + |y^*_{n+1}|)$ instead of $|y^*_{n+1}|$. Why might this be a good idea?

1.2.2 Methods with Memory

Adams Methods

On reaching t_n, we generally have available previously computed solution values $y_n, y_{n-1}, \ldots$ and slopes $f_n = f(t_n, y_n), f_{n-1} = f(t_{n-1}, y_{n-1}), \ldots$ that might be used in computing y_{n+1}. A natural way to exploit this information is a variation on the quadrature approach of the last section. Recall that to approximate the local solution defined by

$$u' = f(t,u), \qquad u(t_n) = y_n$$

we integrated to obtain

$$u(t_n + h) = y_n + \int_{t_n}^{t_n+h} f(x, u(x))\, dx$$

Interpolating s values $f_{n,j} = f(t_{n,j}, y_{n,j})$ with a polynomial $P(x)$ and integrating led to a formula of the form

$$y_{n+1} = y_n + h\sum_{j=1}^{s} A_j f(t_{n,j}, y_{n,j})$$

We found that if the values $y_{n,j} \approx u(t_{n,j})$ are sufficiently accurate, this formula has order at least s. For one–step methods we required that the points $t_{n,j} \in [t_n, t_n + h]$. The question then was how to compute sufficiently accurate $y_{n,j}$. A natural alternative is to take $t_{n,j} = t_{n-j}$ because generally we already have sufficiently accurate approximations $y_{n,j} = y_{n-j}$ and more usefully, $f_{n-j} = f(t_{n,j}, y_{n,j})$. This choice results in a family of explicit formulas called the *Adams–Bashforth formulas*. The lowest order formula in this family is the forward Euler formula because it is the result of interpolating

f_n alone. In the present context it is called AB1. Interpolating f_n and f_{n-1} results after a little calculation in the second order formula, AB2,

$$y_{n+1} = y_n + h_n \left[\left(1 + \frac{r}{2}\right) f_n - \left(\frac{r}{2}\right) f_{n-1} \right]$$

where $r = \frac{h_n}{h_{n-1}}$ (see Exercise 1.5). Note, because we use previously computed values, the mesh spacing in the span of the memory appears explicitly in the coefficients. This is true in general, so it is necessary at each step to work out the coefficients of the formula. Techniques have been devised for doing this efficiently. For theoretical purposes such formulas are often studied with the assumption that the step size is a constant h, in which case $r = 1$ and the formula for AB2 simplifies to

$$y_{n+1} = y_n + h \left[\frac{3}{2} f_n - \frac{1}{2} f_{n-1} \right]$$

The *Adams–Moulton formulas* arise in the same way except that the polynomial interpolates f_{n+1}. Because $f_{n+1} = f(t_{n+1}, y_{n+1})$ involves y_{n+1}, these formulas are implicit. The backward Euler method is the formula of order 1 and the trapezoidal rule, the formula of order 2. In this context they are called AM1 and AM2, respectively.

The accuracy of the Adams methods can be analyzed much as we did with Runge–Kutta methods. It turns out that the (implicit) Adams–Moulton formula of order k, AMk, is more accurate and more stable than the corresponding (explicit) Adams–Bashforth formula of order k, ABk. Which is preferred depends on how much it costs to solve the nonlinear algebraic equation associated with the implicit formula. For non–stiff problems implicit formulas are evaluated by what is called *simple iteration*. We'll illustrate the iteration with the concrete example of AM1 because the general case is exactly the same and there are fewer terms to distract us. We solve iteratively the algebraic equations

$$y_{n+1} = y_n + hf(t_{n+1}, y_{n+1}) \tag{1.23}$$

for y_{n+1}. If the current iterate is $y_{n+1}^{[m]}$, the next is given by

$$y_{n+1}^{[m+1]} = y_n + hf(t_{n+1}, y_{n+1}^{[m]})$$

This operation is described as "correcting" the current iterate with an implicit formula that is called a *corrector formula*. Supposing that the algebraic equations (1.23) have a solution, it follows easily from the Lipschitz condition on f that

$$\|y_{n+1} - y_{n+1}^{[m+1]}\| = \|hf(t_{n+1}, y_{n+1}) - hf(t_{n+1}, y_{n+1}^{[m]})\| \leq hL\|y_{n+1} - y_{n+1}^{[m]}\|$$

From this we see that if $hL < 1$, the new iterate is closer to y_{n+1} than the previous iterate and eventually we have convergence. This argument can be refined to prove that for all sufficiently small step sizes h, the algebraic equations (1.23) have a solution and it is unique. (This is an example of a fixed–point argument that is common in applied mathematics.) The smaller the value of h, the faster this iteration converges. This is important because each iteration costs an evaluation of the ODE function f and if many iterates are necessary, we might just as well use an explicit method with a smaller step size to achieve the same accuracy. On the other hand, for the sake of efficiency we want to use the largest step size that we can. An important way to reduce the number of iterations is to make a good initial guess $y_{n+1}^{[0]}$ for y_{n+1}. There is an easy and natural way to do this—predict the new value $y_{n+1}^{[0]}$ using an explicit formula, a *predictor formula*. For the implicit Adams–Moulton formula AMk, a natural predictor is an explicit Adams–Bashforth formula, either ABk or AB(k-1). Another important way to reduce the cost is to recognize that in practice it is not necessary to evaluate the implicit formula exactly, just well enough that the accuracy of the integration is not impaired. With considerable art in the implementation, a modern code like `VODE` [19] that uses Adams–Moulton methods for non–stiff problems averages about two evaluations of f per step. Simple iteration is practical for non–stiff problems because in the classical situation that $L(b-a)$ is not large, the requirement $Lh < 1$ cannot restrict the step size greatly. Choosing the

step size to compute an accurate solution usually restricts the step size more than enough to ensure the rapid convergence of simple iteration.

There is an important variant of Adams methods that exemplifies a class of methods called *predictor–corrector methods*. A prediction is made with an Adams–Bashforth formula and then a *fixed* number of corrections is made with a corresponding Adams–Moulton formula. The most widely used methods of this kind correct only once. As it happens, Heun's method (1.18) is an example. The value $y_{n+1}^{[0]}$ is predicted with Euler's method, AB1. As we wrote Heun's method earlier, this value was called $y_{n,1}$. The rest of Heun's method is recognized as one correction with the trapezoidal rule, AM2, to give y_{n+1}, followed by evaluating $f(t_{n+1}, y_{n+1})$ for use on the next step. Proceeding in this way gives a *PECE* method (Predict–Evaluate f-Correct-Evaluate f). It would be natural to predict for AM2 with AB2, but this is not necessary and there are some advantages to using the lower order predictor. It is not hard to show that predicting with a formula of order $k-1$ and correcting once with a formula of order k results in a predictor–corrector formula of order k. The argument is much like the one used earlier for quadrature in general and Heun's method in particular. Although the predictor–corrector formula has the same order as the implicit formula used as corrector, the leading term in an expansion of the error is different. Exercise 1.7 provides an example of this. It is important to understand that a predictor–corrector method is an *explicit* method. Accordingly, it has qualitative properties that are in some respects quite different from the *implicit* formula that is used as the corrector. We'll mention one when we discuss stability. These different kinds of formulas are commonly confused because for non–stiff problems, implicit formulas are evaluated by a prediction and correction process. The practical distinction is whether you use as many iterations as necessary to evaluate the implicit formula to a specified accuracy or use a fixed number of iterations. A difficulty with implicit methods is deciding reliably when the iteration to solve the formula has converged to sufficient accuracy, an issue not present with the explicit predictor–corrector methods. Although the two kinds of methods can differ substantially in certain situations, an Adams code that implements Adams methods as predictor–corrector pairs such as `ODE/STEP,INTRP` [115] or `ode113` performs much like an implementation of the Adams–Moulton implicit formulas such as in `DIFSUB` [45] or in `VODE` [19]. The practice that you get implementing such formulas in Exercise 1.10 will help you understand better the distinction between predictor–corrector methods and implicit methods.

BDF methods

On reaching t_n, the *backward differentiation formula* of order k (BDFk) approximates the solution $y(t)$ by the polynomial $P(t)$ that interpolates y_{n+1} and the previously computed approximations $y_n, y_{n-1}, \ldots, y_{n+1-k}$. The polynomial is to satisfy the ODE at t_{n+1}, or in a terminology that will be important for BVPs, it is to *collocate* the ODE at t_{n+1}. This requirement amounts to an algebraic equation for y_{n+1},

$$P'(t_{n+1}) = f(t_{n+1}, P(t_{n+1})) = f(t_{n+1}, y_{n+1})$$

BDF1 results from linear interpolation at t_{n+1} and t_n. The interpolating polynomial has a constant derivative, so the collocation equation is seen immediately to be

$$\frac{y_{n+1} - y_n}{h_n} = f(t_{n+1}, y_{n+1})$$

or equivalently,

$$y_{n+1} - y_n = h_n f(t_{n+1}, y_{n+1})$$

Interpolating with a quadratic polynomial at the three points t_{n+1}, t_n, and t_{n-1} leads in the same way to BDF2,

$$\left(\frac{1+2r}{1+r}\right) y_{n+1} - (1+r)\, y_n + \left(\frac{r^2}{1+r}\right) y_{n-1} = h_n f(t_{n+1}, y_{n+1})$$

where $r = \frac{h_n}{h_{n-1}}$ (see Exercise 1.5). For a reason that we'll take up shortly, the BDFs are generally used with a constant step size h for a number of steps. In this case, BDF2 is

$$\frac{3}{2} y_{n+1} - 2y_n + \frac{1}{2} y_{n-1} = h f(t_{n+1}, y_{n+1})$$

When the step size is a constant h, the Adams formulas and the BDFs are members of a class of formulas called *linear multistep methods* (LMMs). These formulas have the form

$$\sum_{i=0}^{k} \alpha_i y_{n+1-i} = h \sum_{i=0}^{k} \beta_i f(t_{n+1-i}, y_{n+1-i}) \tag{1.24}$$

In proving convergence for one–step methods, the propagation of the error made in a step could be bounded in terms of the stability of the IVP as the maximum step size tends to zero. Convergence is harder to prove when there is a memory because the error made in the current step depends much more strongly on the error made in preceding steps. This requires a shift of focus from approximating local solutions to approximating the global solution $y(t)$ and from the stability of the IVP to that of the numerical method. The *discretization error* or *local truncation error*, lte_n, of a linear multistep method is defined by

$$\begin{aligned} lte_n &= \sum_{i=0}^{k} \alpha_i y(t_{n+1-i}) - h \sum_{i=0}^{k} \beta_i f(t_{n+1-i}, y(t_{n+1-i})) \\ &= \sum_{i=0}^{k} \alpha_i y(t_{n+1-i}) - h \sum_{i=0}^{k} \beta_i y'(t_{n+1-i}) \end{aligned}$$

A straightforward expansion of the terms in Taylor series about t_{n+1} shows that

$$lte_n = \sum_{j=0}^{\infty} C_j h^j y^{(j)}(t_{n+1}) \tag{1.25}$$

where

$$\begin{aligned} C_0 &= \sum_{i=0}^{k} \alpha_i, \quad C_1 = -\sum_{i=0}^{k} \left[i\,\alpha_i + \beta_i \right], \\ C_j &= (-1)^j \sum_{i=1}^{k} \left[\frac{i^j \alpha_i}{j!} + \frac{i^{j-1} \beta_i}{(j-1)!} \right], j = 2, 3, \ldots \end{aligned}$$

If an LMM is convergent, it is of order p when its local truncation error is $O(h^{p+1})$. The expansion (1.25) shows that the formula is of order p when $C_j = 0$ for $j = 0, 1, \ldots, p$. Furthermore,

$$lte_n = C_{p+1} h^{p+1} y^{(p+1)}(t_{n+1}) + \ldots = C_{p+1} h^{p+1} y^{(p+1)}(t_n) + \ldots \tag{1.26}$$

A couple of simple examples will be useful. Rearranging the Taylor series expansion

$$y(t_n) = y(t_{n+1}) - h y'(t_{n+1}) + \frac{h^2}{2} y''(t_{n+1}) + \ldots$$

provides a direct proof that the local truncation error of the backward Euler method is

$$lte_n = -\frac{h^2}{2} y''(t_n + h) + \ldots = -\frac{h^2}{2} y''(t_n) + \ldots \tag{1.27}$$

We leave for Exercise 1.6 the computation that shows the local truncation error of the trapezoidal rule, AM2, is

$$lte_n = -\frac{h^3}{12} y'''(t_n) + \ldots \tag{1.28}$$

A few details about proving convergence will prove illuminating. Suppose that we have an explicit LMM of the form

$$y_{n+1} = y_n + h \sum_{i=1}^{k} \beta_i f(t_{n+1-i}, y_{n+1-i})$$

a form that includes the Adams–Bashforth formulas. The solution of the ODE satisfies this equation with a small perturbation, the local truncation error,

$$y(t_{n+1}) = y(t_n) + h\sum_{i=1}^{k} \beta_i f(t_{n+1-i}, y(t_{n+1-i})) + lte_n$$

Subtracting the first equation from the second results in

$$y(t_{n+1}) - y_{n+1} = y(t_n) - y_n + h\sum_{i=1}^{k} \beta_i [f(t_{n+1-i}, y(t_{n+1-i})) - f(t_{n+1-i}, y_{n+1-i}] + lte_n$$

If the local truncation error is $O(h^{p+1})$, we can take norms and use the Lipschitz condition to obtain

$$\|y(t_{n+1}) - y_{n+1}\| \leq \|y(t_n) - y_n\| + h\sum_{i=1}^{k} |\beta_i| L \|y(t_{n+1-i}) - y_{n+1-i}\| + Ch^{p+1}$$

This inequality involves errors at steps prior to t_n. The trick to dealing with them is to let

$$E_m = \max_{j \leq m} \|y(t_j) - y_j\|$$

Using this quantity in the inequality, we have

$$\|y(t_{n+1}) - y_{n+1}\| \leq (1 + h\mathcal{L})\, E_n + Ch^{p+1}$$

where

$$L\sum_{i=0}^{k} |\beta_i| = \mathcal{L}$$

It then follows that

$$E_{n+1} \leq (1 + h\mathcal{L})\, E_n + Ch^{p+1}$$

This bound on the growth of the error in one step is just like the one we saw earlier for one–step methods. It is now easy to go on to prove that the error on the whole interval $[a, b]$ is $O(h^p)$. A small modification of the argument proves convergence for implicit methods that have the form of Adams–Moulton formulas. A similar argument can be used to prove convergence of predictor–corrector pairs like AB–AM in PECE form.

In the convergence proof just sketched, we bounded directly the error, but often convergence is proven by first showing that the effects of small perturbations to the numerical solution are not amplified by more than a factor of $O(h^{-1})$. Because it takes $(b-a)h^{-1}$ steps to integrate from a to b, we can think of this as stating roughly that the errors do no more than add up. A formula with this property is said to be *zero–stable*. At mesh points the solution of the ODE satisfies the formula with a small perturbation, the local truncation error, that is $O(h^{p+1})$. Zero–stability then implies that the difference between the solution at mesh points and the numerical solution is $O(h^p)$, i.e., the method is convergent and of order p.

A convergence proof for general LMMs starts off in the same way, but now errors at previous steps appear without being multiplied by a factor of h. This makes the analysis more difficult, but more important, for some formulas the errors at previous steps are amplified so much that the method does not converge. The classical theory of LMMs, see e.g. [56, 57], provides conditions on the coefficients of a formula that are necessary and sufficient for zero–stability. It turns out that zero–stable LMMs can have only about half the order of accuracy that is possible for the data used. One reason for giving our attention to the Adams formulas and BDFs is that they are zero–stable and have about the highest order that is possible. The restriction on the order is necessary for a formula to be stable, but it is not sufficient. In fact, the BDFs of orders 7 and up are not zero–stable.

An example [65, p. 381] of a linear multistep method that is not zero–stable is the explicit third order formula

$$y_{n+1} + \frac{3}{2}y_n - 3y_{n-1} + \frac{1}{2}y_{n-2} - 3hf(t_n, y_n) = 0 \tag{1.29}$$

In a numerical experiment we contrast it with AB3,

$$y_{n+1} - y_n - h\left[\frac{23}{12}f(t_n, y_n) - \frac{16}{12}f(t_{n-1}, y_{n-1}) + \frac{5}{12}f(t_{n-2}, y_{n-2})\right] = 0 \tag{1.30}$$

These formulas use the same data and have the same order, so it is not immediately obvious that one is useful and the other is not. Nevertheless, the theory of LMMs tells us that as $h \to 0$, the zero–stable formula AB3 is convergent and formula (1.29) is not. The numerical experiment is to study how the maximum error depends on the step size h when integrating the IVP $y' = -y$, $y(0) = 1$ over $[0, 1]$. Starting values are taken from the analytical solution. Table 1.4 displays the maximum error for step sizes $h = 2^{-i}, i = 2, 3, \ldots, 10$. It appears that AB3 is converging as $h \to 0$. Indeed, when h is halved, the maximum error is divided by about 8, as we might expect of a third order formula. It appears that formula (1.29) is not converging. The error is not so bad for the larger values of h because only a few steps are taken after starting with exact values, but not many steps are needed for the error to be amplified to the extent that the approximate solutions are unacceptable. Exercise 1.8 asks you first to verify that the formula (1.29) is of order three and then to do this experiment yourself.

i	AB3	Formula (1.29)
2	1.34e−003	9.68e−004
3	2.31e−004	6.16e−003
4	3.15e−005	1.27e+000
5	4.08e−006	6.43e+005
6	5.18e−007	2.27e+018
7	6.53e−008	4.23e+044
8	8.19e−009	2.27e+098
9	1.03e−009	1.03e+207
10	1.28e−010	Inf

Table 1.4: Maximum error when $h = 2^{-i}$.

When the step size is varied, we must expect some restrictions on how fast it can change if we are to have stability and convergence as a maximum step size tends to zero. If the ratio of successive step sizes is uniformly bounded above, stability and convergence can be proved for Adams methods much as we did for constant step size [110]. In practice this condition is satisfied because the solvers limit the rate of increase of step size for reasons explained in §1.2.1. The BDFs are another matter. The classical theory for LMMs shows that the BDFs of order less than 7 are stable and convergent as a constant step size tends to zero. They are often implemented so that the solver uses a constant step size until it appears to be advantageous to change to a different constant step size. There are theoretical results that show stability and convergence of such an implementation provided that the changes of step size are limited in size and frequency. Unfortunately, the situation remains murky because the theoretical results do not account for changes as large and frequent as those seen in practice.

The numerical methods that are used in practice are all stable as a maximum step size tends to zero. But in a specific integration, are the step sizes small enough that the integration is stable? It is easy enough to write down expressions for the propagation of small perturbations to a numerical solution, but they are so complicated that it is difficult to gain any insight. For guidance we turn to a standard analysis of the stability of the ODE itself. The idea is to approximate an equation in autonomous form, $y' = f(y)$, near a point (t^*, y^*) by a linear equation with constant coefficients,

$$u' = f(y^*) + f_y(y^*)(u - y^*) \tag{1.31}$$

The approximating ODE is unstable if there are solutions starting near (t^*, y^*) that spread apart rapidly. The difference of any two solutions of this linear equation is a solution of the homogeneous equation

$$v' = f_y(y^*)v$$

For simplicity it is usual to assume that the local Jacobian $f_y(y^*)$ is diagonalizable, meaning that there is a matrix of eigenvectors T such that $T^{-1}f_y(y^*)T = diag\{\lambda_1, \lambda_2, \ldots, \lambda_d\}$. If we change variables to $w = T^{-1}v$, we find that the equations uncouple: Component j of w satisfies $w_j' = \lambda_j w_j$. Each of these equations is an example of the (scalar) *test equation*

$$w' = \lambda w \tag{1.32}$$

with λ an eigenvalue of the local Jacobian. Solving these equations we find that if $Re(\lambda_j) > 0$ for some j, then $w_j(t)$ grows exponentially fast and so does $v(t)$. Because the difference between two solutions of (1.31) grows exponentially fast, the ODE is unstable. On the other hand, if $Re(\lambda_j) \leq 0$ for all j, $w(t)$ is bounded and so is $v(t)$. The approximating equation (1.31) is then stable near (t^*, y^*). The standard numerical methods can be analyzed in a similar way to find that the stability of the method depends on the numerical solution of the test equation. As we shall see, it is not hard to work out the behavior of the numerical solution of this equation when the step size is a constant h. With it we can answer the fundamental question, if $Re(\lambda_j) \leq 0$ for all j so that the approximating ODE is stable near (t^*, y^*), how small does h have to be for the numerical solution to be stable? The theory of *absolute stability* that we have outlined involves a good many approximations, but it has proven to be quite helpful in understanding practical computation. We can also think of it as providing a necessary condition, for if a method does not do a good job of solving the test equation, it can be of only limited value for general equations.

Euler's method provides a simple example of the theory of absolute stability. Suppose that $Re(\lambda) \leq 0$ so that all solutions of the test equation are bounded and the equation is stable. If we apply the forward Euler method to this equation, we find that

$$y_{n+1} = y_n + h\lambda y_n = (1 + h\lambda)y_n$$

Clearly it is necessary that $|1 + h\lambda| \leq 1$ for a bounded numerical solution. The set

$$S = \{|1 + z| \leq 1, Re(z) \leq 0\}$$

is called the *(absolute) stability region* of the forward Euler method. If $h\lambda \in S$, the numerical method is stable, just like the differential equation. If $h\lambda$ is not in S, the integration blows up. This is applied to more general problems by requiring that $h\lambda_j \in S$ for all the eigenvalues λ_j of the local Jacobian f_y. Again we caution that a good many approximations are made in the theory of absolute stability. Nevertheless, experience tells us that it provides valuable insight.

Whatever the value of λ, the forward Euler method is stable for the test equation for all sufficiently small h, as we already knew because the method converges. Notice that $L = |\lambda|$ is a Lipschitz constant for the test equation. In the classical situation that $L(b-a)$ is not large, any restriction on the step size necessary to keep the computation stable cannot be severe. But what if $Re(\lambda) < 0$ and $|\lambda|(b-a) \gg 1$? In this case, the differential equation is stable, indeed extremely so because solutions approach one another exponentially fast, but stability of the forward Euler method requires that h be not much larger than $|\lambda|^{-1}$.

The step size that might be used in practice is mainly determined by two criteria, accuracy and stability. Generally the step size is determined by the accuracy of a formula, but for *stiff* problems, it may be determined by stability. To understand better what stiffness is, let us consider the solution of the IVP

$$y' = -100y + 10, \qquad y(0) = 1 \tag{1.33}$$

on, say, $0 \leq t \leq 10$. Exercise 1.11 asks you to determine the largest step size for which the leading term of the local truncation error of the forward Euler method is smaller in magnitude than a specified absolute error, but the qualitative behavior of the step size is clear. The analytical solution

$$y(t) = \frac{1}{10} + \frac{9}{10}\, e^{-100t}$$

shows that there is an initial period of very rapid change called a boundary layer or initial transient. In this region the method must use a small step size to approximate the solution accurately, a step

size so small that the integration is automatically stable. The problem is not stiff for this method in the initial transient. After a short time t, the solution $y(t)$ is very nearly constant, so an accurate solution can be obtained with a large step size. However, the step size must satisfy $|1+h(-100)| \leq 1$ if the computation is to be stable. This is frustrating. The solution is easy to approximate, but if we try a step size larger than $h = 0.02$, the integration will blow up. Modern codes that select the step size automatically do not blow up: If the step size is small enough that the integration is stable, the code increases the step size because the solution is easy to approximate. When the step size is increased to the point that it is too large for stability, errors begin to grow. When the error becomes too large, the step is a failure and the step size is reduced. This is repeated until the step size is small enough that the integration is again stable and the cycle repeats. The code computes an accurate solution, but the computation is expensive not only because a small step size is necessary for stability, but also because there are many failed steps. Exercise 1.15 provides an example.

A numerical experiment with the IVP (1.33) is instructive. Because $0 < y(t) \leq 1$, a relative error tolerance of 10^{-12} and a modest absolute error tolerance is effectively a pure absolute error control. Table 1.5 shows what happens when we solve this IVP with the explicit Runge–Kutta code `ode45` for a range of absolute error tolerances. This code is intended for non–stiff problems, but it can solve the IVP because its error control keeps the integration stable. Further, the cost of solving the IVP is tolerable because it is only mildly stiff. Notice that there is a relatively large number of failed steps. Also, the number of successful steps is roughly constant as the tolerance is decreased. This is characteristic of stiff problems because for much of the integration, the step size is not determined by accuracy. The BDF code `ode15s` that is intended for stiff problems behaves quite differently. The number of successful steps depends on the tolerance because the step size is determined by accuracy. For this problem, `ode15s` does not have the step failures that `ode45` does because of its finite stability region.

solver	AE	ME	SS	FS
ode45	1e-1	1.1e-1	303	26
	1e-2	1.1e-2	304	26
	1e-3	1.1e-3	307	19
	1e-4	1.1e-4	309	19
ode15s	1e-1	3.4e-2	23	0
	1e-2	8.2e-3	29	0
	1e-3	1.1e-3	39	0
	1e-4	1.6e-4	65	0

Table 1.5: Solution of a mildly stiff IVP: AE–absolute error tolerance, ME–maximum error, SS–number of successful steps, FS–number of failed steps.

Stiff problems are very important in practice, so we must look for methods more stable than the forward Euler method. We don't need to look far. If we solve the test equation with the backward Euler method, the formula is

$$y_{n+1} = y_n + h\lambda y_{n+1}$$

hence

$$y_{n+1} = \frac{1}{1 - h\lambda} y_n \tag{1.34}$$

The stability region of the backward Euler method is then the set

$$S = \left\{ \left| \frac{1}{1-z} \right| \leq 1, Re(z) \leq 0 \right\}$$

which is found to be the whole left half of the complex plane. This is a property called *A–stability*. There appears to be no restriction on the step size for a stable integration with the backward Euler method. Many approximations were made in the stability analysis, so you shouldn't take this conclusion too seriously. That said, the backward Euler method does have excellent stability. This

method is BDF1 and similarly, the BDFs of orders 2 through 6 have stability regions that extend to infinity. Only the formulas of orders 1 and 2 are A–stable. The others are stable in a sector

$$\left\{z = re^{i\theta} \,|\, r > 0, \pi - \alpha < \theta < \pi + \alpha\right\}$$

that contains the entire negative real axis. This restricted version of A–stability is called *$A(\alpha)$–stability*. As the order increases, the angle α becomes smaller and BDF7 is not stable for any angle α; that is, it is not stable at all.

Like BDF1, the trapezoidal rule is A–stable. The predictor–corrector pair that consists of the forward Euler predictor and one correction with the trapezoidal rule is an explicit Runge–Kutta formula, Heun's method. It is not hard to work out the stability region of Heun's method and so learn that it is finite. The fact is, *all* explicit Runge–Kutta methods have finite stability regions. This shows that an implicit method and a predictor–corrector pair using the same implicit method as a corrector can have important qualitative differences.

Exercise 1.9 takes up stability regions for some one–step methods and asks you to prove some of the facts just stated. Exercise 1.16 takes up the computation of stability regions for LMMs. For stability all we ask is that perturbations of the numerical solution not grow. However, if $Re(\lambda) < 0$, perturbations of the ODE itself decay exponentially fast. It would be nice if the numerical method had a similar behavior. The expression (1.34) for BDF1 shows that perturbations are damped strongly when $hRe(\lambda) << -1$. An $A(\alpha)$-stable formula with this desirable property is said to be *$L(\alpha)$–stable*. An attractive feature is that the convergent BDFs are $L(\alpha)$–stable. For the trapezoidal rule, perturbations are barely damped when $hRe(\lambda) << -1$; the formula is A–stable, but not $L(\alpha)$–stable.

Interestingly, the BDFs are stable for *all* sufficiently large $|h\lambda|$. If $Re(\lambda) > 0$ and a BDF is stable for $h\lambda$, the solution of the test equation grows and the numerical solution decays. Of course the numerical solution has the right qualitative behavior for all sufficiently small $h\lambda$ because the method is convergent, but if $Re(\lambda) > 0$ and h is too large, you may not like the heavy damping of this formula. Exercises 1.12 and 1.13 take up implications of this. Exercise 1.14 considers how the implementation affects stability.

What's the catch? Why have we even been talking about methods for non–stiff problems? The answer has two parts. One is that all the methods with infinite stability regions are implicit. Earlier when we derived the backward Euler method as AM1, we talked about evaluating it by simple iteration. Unfortunately, the restriction on the step size for convergence of simple iteration is every bit as severe as that due to stability for an explicit method. To see why this is so, let us evaluate the backward Euler method by simple iteration when solving the test equation. The iteration is

$$y_{n+1}^{[m+1]} = y_n + h\lambda y_{n+1}^{[m]}$$

and

$$|y_{n+1} - y_{n+1}^{[m+1]}| = |h\lambda||y_{n+1} - y_{n+1}^{[m]}|$$

Clearly we must have $|h\lambda| < 1$ for convergence. This is not important when solving non–stiff problems, but it is not acceptable when solving stiff problems. For the example (1.33) of a stiff IVP, h must be less than 0.01 for convergence. This restriction of the step size due to simple iteration when using the backward Euler method is worse than the one due to stability when using the forward Euler method! To solve stiff problems we must resort to a more powerful way of solving the algebraic equations of an implicit method. In the case of the BDFs, these equations have the form

$$y_{n+1} = h\gamma f(t_{n+1}, y_{n+1}) + \psi \tag{1.35}$$

Here γ is a constant that is characteristic of the method and ψ lumps together terms involving the memory $y_n, y_{n-1}, \ldots$. As with simple iteration for non–stiff problems, it is important to make a good initial guess $y_{n+1}^{[0]}$. This can be achieved by interpolating $y_n, y_{n-1}, \ldots$ with a polynomial $Q(t)$ and taking $y_{n+1}^{[0]} = Q(t_{n+1})$. A simplified Newton (chord) method can then be used to solve the algebraic equations iteratively. The equations are linearized approximately as

$$y_{n+1}^{[m+1]} = \psi + h\gamma\left[f(t_{n+1}, y_{n+1}^{[m]}) + J(y_{n+1}^{[m+1]} - y_{n+1}^{[m]})\right]$$

Here

$$J \approx \frac{\partial f}{\partial y}(t_{n+1}, y_{n+1})$$

This way of writing the iteration shows clearly the approximate linearization, but it is very important to organize the computation properly. The next iterate should be computed as a correction Δ_m to the current iterate. A little manipulation shows that

$$\begin{aligned} (I - h\gamma J)\Delta_m &= \psi + h\gamma f(t_{n+1}, y_{n+1}^{[m]}) - y_{n+1}^{[m]} \\ y_{n+1}^{[m+1]} &= y_{n+1}^{[m]} + \Delta_m \end{aligned} \tag{1.36}$$

The iteration matrix $I - h\gamma J$ is very ill–conditioned when the IVP is stiff. Exercise 1.20 explains how to modify `ode15s` so that you can monitor the condition of the iteration matrix in the course of an integration and asks you verify this assertion for one IVP. When solving a very ill–conditioned linear system, it may be that only a few of the leading digits of the solution are computed correctly. Accordingly, if we try to compute the iterates $y_{n+1}^{[m+1]}$ directly, we may not be able to compute more than a few digits correctly. However, if we compute a few correct digits in each increment Δ_m, more and more digits will be correct in the iterates $y_{n+1}^{[m+1]}$. Notice that the right hand side of the linear system (1.36) is the residual of the current iterate. It is a measure of how well the current iterate satisfies the algebraic equations (1.35). By holding fixed the approximation J to the local Jacobian, an LU factorization (that is, an LU decomposition) of the iteration matrix $I - h_n\gamma J$ can be computed and then used to solve efficiently all the linear systems of the iteration at the point t_{n+1}. Forming and factoring $I - h_n\gamma J$ is relatively expensive, so BDF codes typically hold the step size constant and use a single LU factorization for several steps. Only when the iterates do not converge sufficiently fast or when it is predicted that a considerably larger step size might be used (perhaps with a change of order, that is, with a different formula) is a new iteration matrix formed and factored.

Evaluating an implicit formula by solving the equation with a simplified Newton iteration can be expensive. In the first place we must form and store an approximation J to the local Jacobian. We then must solve a linear system with matrix $I - h\gamma J$ for each iterate. It can be expensive to approximate Jacobians, so the codes try to minimize the number of times this is done. A new iteration matrix must be formed when the step size changes significantly. The early codes for stiff IVPs formed a new approximation to the Jacobian then and overwrote it with the new iteration matrix because storage was at a premium. This is still appropriate when solving extremely large systems, but nowadays some solvers for stiff IVPs, including those of MATLAB, save J and reuse it in the iteration matrix for as long as the iteration converges at an acceptable rate. This implies that if such a solver is applied to a problem that is not stiff, very few Jacobians are formed. Along with the fast linear algebra of the MATLAB PSE, this makes the stiff solvers of MATLAB reasonably efficient for non–stiff problems of modest size.

Because it is so convenient, all the codes for stiff IVPs have an option for approximating Jacobians by finite differences. This is the default option for the MATLAB solvers. Suppose that we want to approximate the matrix $f_y(t^*, y^*)$. If $e^{(j)}$ is column j of the identity matrix, the vector $y^* + \delta_j e^{(j)}$ represents a change of δ_j in component j of y^*. From the Taylor series

$$f(t^*, y^* + \delta e^{(j)}) = f(t^*, y^*) + f_y(t^*, y^*)\delta_j e^{(j)} + O(\delta_j^2)$$

we obtain an approximation to column j of the Jacobian:

$$f_y(t^*, y^*)e^{(j)} \approx \delta_j^{-1}\left[f(t^*, y^* + \delta e^{(j)}) - f(t^*, y^*)\right]$$

For a system of d equations, we can obtain an approximation to the Jacobian in this way by making d evaluations of f. This is the standard way of approximating Jacobians, but the algorithms differ considerably in detail because it is hard to choose a good value for δ_j. It must be small enough that the finite difference provide a good approximation to the partial derivatives, but not so small that the approximation consist only of roundoff error. When the components of f differ greatly in size,

it may not even be possible to find one value of δ_j that is good for all components of the vector of partial derivatives. Fortunately, we do not need an accurate Jacobian, just an approximation that is good enough to achive acceptable convergence in the simplified Newton iteration. A few algorithms, including the `numjac` function of MATLAB, monitor the differences in the function values. They adjust the sizes of the increments δ_j based on experience in approximating a previous Jacobian and repeat the approximation of a column with a different increment when this appears to necessary for an acceptable approximation. Numerical Jacobians are convenient and generally satisfactory, but the solvers are more robust and perhaps faster if you provide an analytical Jacobian.

For a large system of ODEs, it is typical that only a few components of y appear in each equation. If component j of y does not appear in component i of $f(t, y)$, the partial derivative $\frac{\partial f_i}{\partial y_j}$ is zero. If most of the entries of a matrix are zero, the matrix is said to be *sparse*. By storing only the non–zero entries of a sparse Jacobian, storage is reduced from the square of the number of equations d to a modest multiple of d. If the Jacobian is sparse, so is the iteration matrix. As with storage, the cost of solving linear systems by elimination can be reduced dramatically by paying attention to zero entries in the matrix. A clever algorithm of Curtis, Powell, and Reid [30] provides a similar reduction in the cost of approximating a sparse Jacobian. By taking into account the known value of zero for most of the entries in the Jacobian, it is typically possible to approximate all the non–zero entries of several columns at a time. An important special case of a sparse Jacobian is one that has all its non–zero entries located in a band of diagonals. For example, if for all i, the entry $J_{i,j} = 0$ for all j except possibly $j = i - 1$, i, and $i + 1$, we say that the matrix J is tridiagonal and has a band width of 3. If there are m diagonals in the band, only m additional evaluations of f are needed to approximate the Jacobian, *no matter how many equations there are*. It is comparatively easy to obtain the advantages of sparsity when the matrix is banded, so all the popular solvers provide for banded Jacobians. Some, including all the solvers of MATLAB, provide for general sparse Jacobians. These algorithmic developments are crucial to the solution of large systems. §1.3.3 provides more information about this and some examples.

Each step in an implementation of the BDFs that uses a simplified Newton iteration to evaluate the formulas is much more expensive, in both computing time and storage, than taking a step with a method intended for non–stiff problems using either simple iteration or no iteration at all. Accordingly, BDFs evaluated with a simplified Newton iteration are advantageous only when the step size would otherwise be greatly restricted by stability, that is, only when the IVP is stiff. For non–stiff problems the BDFs could be evaluated with simple iteration, but this is rarely done because the Adams–Moulton methods have a similar structure and are considerably more accurate.

It is unfortunate that stiff problems are not easily recognized. We do have a clear distinction between methods and implementations for stiff and non–stiff problems. A practical definition of a stiff problem is that it is a problem such that solving it with a method intended for stiff problems is much more efficient than solving it with a method intended for non–stiff problems. Exercises 1.15, 1.18, 1.34, and 1.36 will give you some experience with this. Insight may provide guidance as to whether a problem is stiff. The ODEs of a stiff problem must allow solutions that change on a scale small compared to the length of the interval of integration. Some physical problems are naturally expressed in terms of time constants, in which case a stiff problem must have some time constant that is small compared to the time interval of interest. The solution of interest must itself be slowly varying; exercise 1.36 makes this point. Most problems that are described as stiff have regions where the solution of interest changes rapidly, such as a boundary layer or an initial transient. In such intervals the problem is not stiff because the solver must use a small step size to represent the solution accurately. The problem is stiff where the solution is easy to approximate and the requirement for stability dominates the choice of step size. Clearly variation of the step size so as to use step sizes appropriate both for the initial transient and for the smooth region is fundamental to the solution of stiff IVPs. The proton transfer and Robertson examples of §**??** illustrate this.

Error Estimation and Change of Order

Estimation of the local truncation error of linear multistep methods is generally considered to be very easy compared to estimation of the local error for Runge–Kutta methods. It *is* easy to derive

estimates—what's hard is to prove that they work. We begin by discussing how the local truncation error might be estimated. With, say, the Adams methods, it is easy to take a step with formulas of two different orders. Indeed, we noted earlier that we could predict with AB(k-1) and obtain a formula of order k with a single correction using AMk. Just as with one–step methods, the error in the formula AB(k-1) of order $k-1$ can be estimated by comparing the result of this formula to the result of order k from the pair. When the size of this error is controlled and the integration is advanced with the more accurate and stable result of the predictor–corrector pair, we are doing local extrapolation. This approach is used in `ode113`. Another approach is based on the expansion (1.26) for the local truncation error when the step size is constant. It shows that the leading terms of the local truncation errors of the Adams–Bashforth and Adams–Moulton methods of the same order differ by a constant multiple. Using this fact, the step is taken with both ABk and AMk and the local truncation error is estimated by an appropriate multiple of the difference of the two results. The more accurate and stable result of AMk is used to advance the integration. The error of this formula is being controlled, so local extrapolation is not done. In this approach the implicit Adams–Moulton methods are evaluated by simple iteration to a specified accuracy. The explicit Adams–Bashforth formula ABk is used to start the iteration for computing AMk and to estimate the local truncation error of AMk. To leading order, the local truncation errors of AMk and the ABk–AMk pair with one iteration are the same, so the same approach to local truncation error estimation can also be used for predictor–corrector implementations.

An approach that is particularly natural for the BDFs is to approximate directly the leading term of the local truncation error. Recall that for BDF1, this is

$$lte_n = -\frac{h^2}{2}y''(t_n) + \ldots$$

The formulas are based on numerical differentiation, so it is natural to interpolate y_{n+1}, y_n, and y_{n-1} with a quadratic polynomial $Q(t)$ and then to use

$$est = -\frac{h^2}{2}Q''(t_n) \approx -\frac{h^2}{2}y''(t_n)$$

This is the way that the local truncation error is estimated for the BDFs in `ode15s`. Similarly, a cubic interpolant is used in `ode23t` to approximate the derivative in the local truncation error (1.28) of the trapezoidal rule. The backward Euler method and the trapezoidal rule are one–step methods, but previously computed approximate solutions are used for estimating local truncation errors. Previously computed solutions are also used to predict the solution of the algebraic equations that must be solved to evaluate these implicit methods. We see from this that in practice the distinction between implicit one–step methods and methods with memory may be blurred.

An interesting and important aspect of these estimates of the local truncation error is that it is possible to estimate the error that would have been made if a different order had been used. The examples of AM1 and AM2 show the possibility. Considering how the local truncation error is estimated, it is clear that when taking a step with the second order formula AM2, we could estimate the local truncation error of the first order formula AM1. At least mechanically, it is also clear that we could use more memorized values to estimate the error that would have been made with the higher order formula AM3. This opens up the possibility of adapting the order (formula) to the solution so as to use larger step sizes. Modern Adams and BDF codes like `ode113` and `ode15s`, respectively, do exactly that. Indeed, the names of these functions indicate which members of the family are used: The Adams code `ode113` uses orders ranging from 1 to 13 and the BDF code `ode15s`, orders ranging from 1 to 5. (The final `s` of `ode15s` indicates that it is intended for stiff IVPs.) Both Adams–Moulton formulas and BDFs are implemented in the one variable order code `DIFSUB` [45] and its descendants such as `VODE` [19]. Variation of the order plays another role in Adams and BDF codes. The lowest order formulas are one–step, so they can be used to take the first step of the integration. Each step of the integration provides another approximate solution value for the memory, so the solver can consider raising the order and increasing the step size for the next step. There are many practical details, but the codes all increase the order rapidly until a value appropriate to the solution has been found. Starting the integration in this way is both

convenient and efficient in a variable step size, variable order (VSVO) implementation, so all the popular Adams and BDF codes do it this way.

Deriving estimates of the local truncation error is easy for methods with memory, but justifying them is hard. The snag is this: When we derived the estimators, we assumed implicitly that the previously computed values and derivatives are exactly equal to the solution values $y(t_n), y(t_{n-1}), \ldots$ and their derivatives $y'(t_n), y'(t_{n-1}), \ldots$, respectively. However, in practice these previously computed values are in error, and these errors may be just as large, or *even larger*, than the error we want to estimate. The difficulty is especially clear when we contemplate estimating the local truncation error of a formula with order higher than the one(s) used to compute the memorized values. Justifying an estimator in realistic circumstances is difficult because it is not even possible without some regularity in the behavior of the error. So far we have discussed only the order of the error. Classic results in the theory of linear multistep methods show that with certain assumptions, the error behaves in a regular way. This regularity can be used to justify error estimation, as was done first by Henrici [56] and then more generally by Stetter [124]. Their results do not apply directly to modern variable order Adams and BDF codes because they assume that the same formula is used all the time and that if the step sizes are varied at all, they are varied in a regular way that is given *a priori*. Little is known about the behavior of the error when the order (formula) is varied in the course of the integration. The theory for constant order provides insight and there are a few results [112] that apply directly, but our theoretical understanding of modern Adams and BDF codes that vary their order leaves much to be desired.

Continuous Extensions

Both Adams methods and BDFs are based on polynomial interpolants, so it is natural to use these interpolants as continuous extensions for the methods. An application of continuous extensions that is special to methods with memory is changing the step size. We have seen that there are practical reasons for working with a constant step size when solving stiff IVPs. An easy way to change to a different constant step size H at t_n is to approximate the solution at $t_n, t_n - H, t_n - 2H, \ldots$ by evaluating the interpolant at these points. With these solution values at a constant mesh spacing of H, we can use a constant step formula with step size H from t_n on. This is a standard technique for changing the step size when integrating with BDFs and some codes use it for Adams methods, too, `DIFSUB` being an example for both kinds of methods.

Exercise 1.5. To understand better the origin of Adams formulas and BDFs

- Work out the details for deriving AB2
- Work out the details for deriving BDF2

Exercise 1.6. Show that the local truncation error of AM2, the trapezoidal rule, is

$$lte_n = -\frac{h^3}{12} y^{(3)}(t_n) + \ldots$$

Exercise 1.7. Show that when solving $y' = -y$, the error made in a step of size h from $(t_n, y(t_n))$ with AM2 (the trapezoidal rule) is

$$y(t_n + h) - y_{n+1} = \frac{1}{12} h^3 y(t_n) + \ldots$$

(This is the local truncation error of AM2 for this ODE.) Show that the corresponding error for AB1–AM2 in PECE form (Heun's method) is

$$y(t_n + h) - y_{n+1} = -\frac{1}{6} h^3 y(t_n) + \ldots$$

Evidently the accuracy of a predictor–corrector method can be different from that of the corrector evaluated as an implicit formula.

Exercise 1.8. Verify that the formula (1.29) is of order three. Do the numerical experiment described in the text that resulted in Table 1.4.

Exercise 1.9. To understand better stability regions

- Show that the stability region of the backward Euler method includes the left half of the complex plane.
- Show that the stability region of the trapezoidal rule is the left half of the complex plane.
- Heun's method can be viewed as a predictor–corrector formula resulting from a prediction with Euler's method and a single correction with the trapezoidal rule. Show that the stability region of Heun's method is finite.

Exercise 1.10. To appreciate better the mechanics of the various kinds of methods, write simple MATLAB programs to solve

$$y' = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} y, \qquad y(0) = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

on $[0, 1]$. Use a constant step size of $h = 0.1$ and plot the numerical solution together with the analytical solution.

- Solve the IVP with Heun's method. For this write a function of the form `[tnp1,ynp1] = Heun(tn,yn,h,f)`. This function is to accept as input the current solution (t_n, y_n), step size h, and the handle f of the function for evaluating the ODEs. It advances the integration to (t_{n+1}, y_{n+1}). You may not be familiar with the way MATLAB evaluates functions that have been passed to another function as an input argument. This is done using the built–in function `feval` as illustrated by a function for taking a step with Euler's method:

  ```
  function [tnp1,ynp1] = Euler(tn,yn,h,f)
  yp = feval(f,tn,yn);
  ynp1 = yn + h*yp;
  tnp1 = tn + h;
  ```

  See the MATLAB documentation for more details about `feval`.

- Solve the IVP with the trapezoidal rule evaluated with simple iteration. Heun's method can be viewed as a predictor–corrector pair resulting from a prediction with Euler's method and a single correction with the trapezoidal rule. Modify the function `Heun` to obtain a function `AM2si` that evaluates the trapezoidal rule by iterating to completion. It is not easy to decide when to stop iterating. For this exercise, if `p` is the current iterate and the new iterate is `c`, accept the new iterate if `norm(c - p) < 1e-3*norm(c)` and otherwise continue iterating. If your iteration does not converge in 10 iterations, terminate the run with an error message.

- This IVP is not stiff, but solve it with the trapezoidal rule to see what is involved in the solution of stiff IVPs. Modify the function `AM2si` so that it has the form `[tnp1,ynp1] = AM2ni(tn,yn,h,f,dfdy)`. Here `dfdy` is the handle of a function $dfdy(t, y)$ for evaluating the Jacobian $f_y(t, y)$. On entry to `AM2ni`, use `feval` to evaluate the (constant) Jacobian J, form the iteration matrix $M = I - 0.5hJ$, and compute its LU decomposition. Use this factorization to compute the iterates. As explained in the text, it is important to code the iteration so that you compute the correction Δ_m and then the new iterate `c` as the sum of the current iterate `p` and the correction. Much as with simple iteration, accept the new iterate if $\|\Delta_m\|$ is less than `1e-3*norm(c)` and otherwise continue iterating. If your iteration does not converge in 10 iterations, terminate the run with an error message. (Newton's method converges immediately for this ODE, but you are to code for general ODEs so as to see what is involved.)

t	y
2.0	0.472
3.0	0.330
4.0	0.248
5.0	0.199
6.0	0.166
7.0	0.142
8.0	0.124
9.0	0.110
10.0	0.099

Table 1.6: Backward Euler solution.

Exercise 1.11. The text discusses the solution of the stiff IVP

$$y' = -100y + 10, \qquad y(0) = 1$$

on $0 \le t \le 10$ with AB1. It is said that a small step size is necessary in the beginning to resolve the solution in a boundary layer, but after a while, the solution is nearly constant and a large step size can be used. Justify this statement by finding the largest step size h for which the magnitude of the leading term in the local truncation error at t_n is no greater than an absolute error tolerance of τ.

Exercise 1.12. For simplicity and convenience, applications codes often use a constant step size when integrating with the backward Euler formula. This formula has very good stability properties, but if they are misused, these properties can lead to a numerical solution that is qualitatively wrong. As a simple example, integrate the IVP

$$y' = 10y, \quad y(0) = 1$$

with the backward Euler method and a step size $h = 1$. How does the numerical solution compare to the analytical solution $y(t) = e^{10t}$? What's going on?

Exercise 1.13. A difficulty related to that of Exercise 1.12 is illustrated by the IVP

$$y' = -\frac{1}{t^2} + 10\left(y - \frac{1}{t}\right), \quad y(1) = 1$$

If we expect on physical grounds that the solution decays to zero, we might be content to approximate it using the backward Euler method and a constant step size. The analytical solution of this problem is $y(t) = t^{-1}$, so it does decay. The results of solving the IVP with step size h =1 displayed in Table 1.6 appear to be satisfactory.

Now solve this IVP with each of the variable step size codes `ode45` and `ode15s`. The numerical solutions are terrible. These are quality solvers, so what is going on? What do these numerical solutions tell you about this IVP?

The difficulty illustrated by this example may not be so obvious when it arises in practice. §1.3.3 discusses how the solution of a PDE might be approximated by the solution of a system of ODEs. There is some art in this. When using finite differences to approximate some kinds of PDEs, it may happen that an "obvious" approximating system of ODEs is unstable. It is possible to damp instabilities by using the backward Euler method and a "large" step size, but whether the numerical solution will then faithfully model the solution of the PDE is problematical.

Exercise 1.14. The backward Euler method is misused in another way in some popular applications codes. It is recognized that the stability of an implicit method is needed, but to keep down the cost, the formula is evaluated by a predictor–corrector process with only one correction. You can't have it both ways: If you want the stability of an implicit method, you must go to the expense of evaluating it properly. To see an example of this, show that the stability region of the predictor–corrector pair

consisting of a prediction with the forward Euler method and one correction with the backward Euler method is finite. The qualitative effect of a single correction is made clear by showing that the predictor–corrector pair is not stable for, say, $z = h\lambda = -2$, whereas the backward Euler method is stable for all z in the left half of the complex plane.

Exercise 1.15. This exercise will help you understand stiffness in both theory and practice. O'Malley [96] models the concentration of a reactant $y(t)$ in a combustion process with the ODE

$$y' = f(y) = y^2(1 - y)$$

that is to be integrated over $[0, 2\epsilon^{-1}]$ with initial condition $y(0) = \epsilon$. He uses perturbation methods to analyze the behavior of the solution for small disturbances $\epsilon > 0$ from the pre–ignition state. The analytical work is illuminating, but for the sake of simplicity, just solve the IVP numerically for $\epsilon = 10^{-4}$ with the solver `ode15s` based on the BDFs and the program

```
function ignition
epsilon = 1e-4;
options = odeset('Stats','on');
ode15s(@ode,[0, 2/epsilon],epsilon,options);

%==================================
function dydt = ode(t,y)
dydt = y^2*(1 - y);
```

This program displays the solution to the screen as it is computed and displays some statistics at the end of the run. You will find that the solution increases slowly from its initial value of ϵ. At a time that is $O(\epsilon^{-1})$, the reactant ignites and increases rapidly to a value near 1. This increase takes place in an interval that is $O(1)$. For the remainder of the interval of integration, the solution is very near to its limit of 1. Modify the program to solve the IVP with the solver `ode45` based on an explicit Runge–Kutta pair—all you must do is change the name of the solver. You will find that the numerical integration stalls after ignition, despite the fact that the solution is very nearly constant then. To quantify the difference, use `tic` and `toc` to measure the run times of the two solvers over the whole interval and over the first half of the interval. In this you should not display the solution as it is computed, i.e., you should change the invocation of `ode15s` to

```
[t,y] = ode15s(@ode,[0, 2/epsilon],epsilon,options);
```

You will find that the non–stiff solver `ode45` is rather faster on the first half of the interval because of the superior accuracy of its formulas, this despite the minimal linear algebra costs in `ode15s` due to the ODE having only one unknown. You will find that the stiff solver `ode15s` is much faster on the whole interval because the explicit Runge–Kutta formulas of `ode45` must use a small step size to keep the integration stable in the last half of the interval and the BDFs of `ode15s` do not. The statistics show that the Runge–Kutta code has many failed steps in the second half of the integration. This is typical when solving a stiff problem with a method that has a finite stability region.

To understand the numerical results, work out the Jacobian f_y. Because there is a single equation, the only eigenvalue is the Jacobian itself. Work out a Lipschitz constant for $0 \le y \le 1$. You will find that it is not large, so the IVP can be stiff only on long intervals. In the first part of the integration the solution is positive, slowly varying, and $O(\epsilon)$. Use the sign of the eigenvalue to argue that the IVP is unstable in a linear stability analysis, hence it is not stiff in this part of the integration. Argue that it is only moderately unstable on this interval of length $O(\epsilon^{-1})$, so we can expect to solve it accurately. The change in the solution at ignition is quite sharp when plotted on $[0, 2\epsilon^{-1}]$, but if you use `zoom`, you will find that most of the change occurs in $[9650, 9680]$. Argue that in an interval of size $O(1)$ like this, the IVP is not stiff. In the last half of the interval of integration the solution is close to 1 and slowly varying. Use the sign of the eigenvalue to argue that the IVP is stable then and argue that the IVP is stiff on this interval of length $O(\epsilon^{-1})$.

Exercise 1.16. The boundary of the stability region for a linear multistep method can be computed by the *root locus method*. An LMM applied to the test equation $y' = \lambda y$ with constant step size h has the form

$$\sum_{i=0}^{k} \alpha_i y_{n+1-i} - h \sum_{i=0}^{k} \beta_i (\lambda y_{n+1-i}) = \sum_{i=0}^{k} (\alpha_i - h\lambda\beta_i)\, y_{n+1-i} = 0$$

This is a linear difference equation of order k with constant coefficients that can be studied much like a linear ODE of order k with constant coefficients. Each root r of the characteristic polynomial

$$\sum_{i=0}^{k} (\alpha_i - h\lambda\beta_i) r^{k-i}$$

provides a solution of the difference equation of the form $y_m = r^m$ for $m = 0, 1, 2, \ldots$. As with ODEs, there are other kinds of solutions when r is a multiple root, but also as with ODEs, it is found that for a given $z = h\lambda$, all solutions of the difference equation are bounded, and therefore the linear multistep method is stable, if all the roots of the characteristic polynomial have magnitude less than 1. For a point z on the boundary of the stability region, a root r has magnitude equal to 1. We can use this observation to trace the boundary of the stability region by plotting all the z for which r is a root of magnitude 1. For this it is convenient to write the polynomial as $\rho(r) - z\sigma(r)$ where

$$\rho(r) = \sum_{i=0}^{k} \alpha_i r^{k-i}, \quad \sigma(r) = \sum_{i=0}^{k} \beta_i r^{k-i}$$

With these definitions, the boundary of the stability region is the curve

$$z = \frac{\rho(r)}{\sigma(r)}$$

for $r = e^{i\theta}$ as θ ranges from 0 to 2π. Plot the boundary of the stability region for several of the linear multistep methods discussed in the text, e.g., the forward Euler method (AB1), the backward Euler method (AM1, BDF1), the trapezoidal rule (AM2), and BDF2. Some other linear multistep formulas that you might try are AM3

$$y_{n+1} = y_n + h\left[\frac{5}{12} y'_{n+1} + \frac{8}{12} y'_n - \frac{1}{12} y'_{n-1}\right]$$

and BDF3

$$y_{n+1} = \frac{18}{11} y_n - \frac{9}{11} y_{n-1} + \frac{2}{11} y_{n-2} + \frac{6}{11} h y'_{n+1}$$

The root locus method just gives you the boundary of the stability region. The BDFs have stability regions that are unbounded, so they are stable *outside* the curve you plot. The trapezoidal rule is a little different because as Exercise 1.9 asks you to prove directly, it is stable in the left half of the complex plane. The other formulas mentioned have finite stability regions, so are stable inside the curve.

1.3 Solving IVPs in Matlab

In the simplest use of the MATLAB solvers, all you must do is tell the solver what IVP is to be solved. That is, you must provide a function that evaluates $f(t, y)$, the interval of integration, and the vector of initial conditions. MATLAB has a good many solvers implementing diverse methods, but they can all used in *exactly* the same way. The documentation suggests that you try `ode45` first unless you suspect that the problem is stiff, in which case you should try `ode15s`. In preceding sections we have discussed what stiffness is, but the practical definition is that if `ode15s` solves the IVP *much* faster than `ode45`, then the problem is stiff. Many people have the impression that if an IVP is hard for a code like `ode45`, it must be stiff. No, it might be hard for a different reason and

so be hard for `ode15s`, too. On the other hand, if a problem is hard for `ode45` and easy for `ode15s`, then you almost certainly have a stiff problem.

There is a programming issue that deserves comment. MATLAB programs can be written as script files or functions. We have preferred to write the examples as functions because then auxiliary functions can be provided as subfunctions. Certainly it is convenient to have the function defining the ODEs available in the same file as the main program. This becomes more important when several functions must be supplied, as when solving IVPs with complications such as event location. Supplying multiple auxiliary functions is always necessary when solving BVPs and DDEs.

Example 1.3.1. In Chapter **??** we looked at the analytical solution of a family of simple equations. (Exercise 1.17 has you solve them numerically.) Among the problems was

$$y' = y^2 + t^2$$

We found the general solution to be a rather complicated expression involving fractional Bessel functions. Still, the equation is simple enough that you can solve analytically for the solution that has $y(0) = 0$ and plot it for $0 \le t \le 1$ at the command line with

```
>> y = dsolve('Dy = y^2 + t^2','y(0) = 0')
>> ezplot(y,[0,1])
```

This way of supplying the ODEs is satisfactory for very simple problems, but the numerical solvers are expected to deal with large and complicated systems of equations. Accordingly, they expect the ODEs to be evaluated with a (sub)function. For this problem, the ODE might be coded as

```
function dydt = f(t,y)
dydt = y^2 + t^2;
```

and saved in the file `f.m`. The IVP is solved numerically on $[0, 1]$ and the solution plotted by the commands

```
[t,y] = ode45(@f,[0,1],0);
plot(t,y)
```

that you code either as a script file or function. Of course, with only two commands, you might as well enter them at the command line. The first input argument tells `ode45` which function is to be used for evaluating the ODEs. This is done with a function handle, here `@f`. The second input argument is the interval of integration $[a, b]$ and the third is the initial value. The solver computes approximations $y_n \approx y(t_n)$ on a mesh

$$a = t_0 < t_1 < \cdots < t_N = b$$

which is chosen by the solver. The mesh points are returned in the array `t` and the corresponding approximate solutions in the array `y`. This output is in a form convenient for plotting. Fig. 1.1 shows the result of this computation. When run as a script (or from the command line) you have available the mesh `t` and the solution `y` on this mesh, so you can, for example, look at individual components of systems or plot with logarithmic scales.

All the input arguments are required even to define the IVP. Indeed, the only argument that might be questioned is the interval of integration. However, we have seen that the stability of an IVP is fundamental to its numerical solution and this depends on both the length of the interval and the direction of integration. The MATLAB problem solving environment and the design of the solvers make it possible to avoid the long call lists of arguments that are typical of solvers written for compiled computation in general scientific computing. For instance, MATLAB makes it possible to avoid the tedious and error–prone specification of storage and data types that are necessary for compiled computation. This is accomplished by making heavy use of default values for optional arguments.

The MATLAB PSE makes convenient the output of solution arrays of a size that cannot be determined in advance and provides convenient tools for plotting these arrays of data. The typical

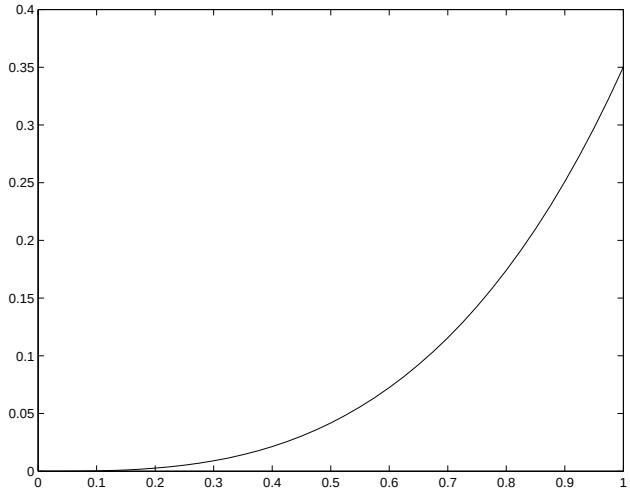

Figure 1.1: Solution of the ODE $y' = y^2 + t^2$ with initial condition $y(0) = 0$.

solver for general scientific computing has you supply all the t_n where you want answers. Storage issues are an important reason for this, but, of course, it may be that you actually want answers at specific points. One way that the MATLAB IVP solvers provide this capability is by overloading the argument that specifies the interval. If you supply an array `[t0 t1 ... tf]` of more than two entries for the interval of integration, the solver interprets this as an instruction to return approximations to the solution values $y(t0), y(t1), \ldots, y(tf)$. When called in this way, the output array `t` is the same as the input array. The fact that the number and placement of specified output points has little effect on the computation is of great practical importance. The output points must be in order, that is, either $a = t0 < t1 < \cdots < tf = b$ or $a = t0 > t1 > \cdots > tf = b$. As a concrete example, suppose that we had wanted to construct a table of the solution of the last example at 10 equally spaced points in the interval $[0, 1]$. This could be achieved with the script

```
tout = linspace(0,1,10);
[t,y] = ode45(@f,tout,0);
```

There is another way to receive output from the IVP solvers that has some advantages. The output can be in the form of a structure that can be given any name, but let us suppose that it is called `sol`. For the present example the syntax is

```
sol = ode45(@f,[0,1],0);
```

In this way of using the solvers, the usual output `[t,y]` is available as fields of the solution structure `sol`. Specifically, the mesh `t` is returned as the field `sol.x` and the solution at these points is returned as the field `sol.y`. However, that is not the point of this form of output. The solvers actually produce a continuous solution $S(t)$ on all of $[a, b]$. They return in the solution structure `sol` the information needed to evaluate $S(t)$. This is done with an auxiliary function `deval`. It has two input arguments, a solution structure and an array of points where you want approximate solutions. Rather than tell a solver to compute answers at `tout = linspace(0,1,10)`, you can have it return a solution structure `sol` and then compute answers at these points with the command

```
Stout = deval(sol,tout);
```

With this mode of output, you solve an IVP just once. Using the solution structure you can then compute answers anywhere you want. In effect, you have a function for the solution. Indeed, to make this more like a function, `deval` also accepts its arguments in the reverse order:

```
Stout = deval(tout,sol);
```

This mode of output is an option for the IVP solvers, but it is the only mode of output from the BVP and DDE solvers of MATLAB.

Some ODEs cannot be integrated past a critical point. How this difficulty is handled depends on the solver. It is easy to deal with the difficulty if the code produces output at a specific point by stepping to that point because you simply do not ask for output beyond the critical point. However, this is an inefficient design. Modern solvers integrate with the largest step sizes they can and evaluate continuous extensions to approximate the solution at the output points. Accordingly, they must be allowed to step past output points. So, what do you do when the solver cannot step past some point? Many codes in general scientific computing have an option for you to inform them of this relatively unusual situation. The MATLAB solvers handle this in a different way. They accept only problems set on an interval. They do not integrate past the end of this interval, nor do they evaluate the ODE at points outside the interval. Accordingly, for this example it does not matter whether the ODE is even defined for $t > 1$.

Example 1.3.2. Example 1.3.1 is unusual in that the IVP is solved from the command line. The program `ch2ex1.m` is more typical. It solves the ODEs

$$\begin{aligned} x_1' &= -k_1 x_1 + k_2 y \\ x_2' &= -k_4 x_2 + k_3 y \\ y' &= k_1 x_1 + k_4 x_2 - (k_1 + k_3) y \end{aligned}$$

with initial values

$$x_1(0) = 0,\ x_2(0) = 1,\ y(0) = 0$$

on the interval $0 \le t \le 8 \cdot 10^5$. The coefficients here are

$$k_1 = 8.4303270 \cdot 10^{-10},\ k_2 = 2.9002673 \cdot 10^{11},\ k_3 = 2.4603642 \cdot 10^{10},\ k_4 = 8.7600580 \cdot 10^{-6}$$

By writing the main program as a function, we can code the evaluation of the ODEs as the subfunction `odes`. The output of this subfunction must be a column vector, which is achieved here by initializing `dydt` as a column vector of zeros. The approximations to $y_i(t_n)$ for $n = 1, 2, \ldots$ are returned as `y(:,i)`. The program uses this output to plot $x_1(t)$ and $x_2(t)$ together with a linear scale for t and plot $y(t)$ separately with a logarithmic scale for t.

This is the proton transfer problem discussed in §**??** where the two plots are provided as Figs. **??** and **??**. It is pointed out there that the default absolute error tolerance of 10^{-6} that is applied to all solution components is inappropriate for $y(t)$ because Fig. **??** shows that it has a magnitude less than $3 \cdot 10^{-17}$. All optional specifications to the MATLAB IVP solvers are provided by means of the auxiliary function `odeset` and *keywords*. The command `help odeset` provides short explanations of the various options and the command `odeset` by itself provides reminders. Error tolerances are the most common optional input. An absolute error tolerance of 10^{-20} is more appropriate for this IVP. When the absolute error tolerance is given a scalar value, it is applied to all solution components. When it is given a vector value, the entries in the vector of absolute error tolerances are applied to corresponding entries in the solution vector. Any option not specified is given its default value. Here we use the default value of 10^{-3} for the relative error tolerance (which is always a scalar), but if we had wanted to assign it a value of, say, 10^{-4}, we would have used the command

```
options = odeset('AbsTol',1e-20,'RelTol',1e-4)
```

Options can be assigned in any order and the keywords are not case sensitive. After forming an options structure with `odeset`, it is provided as the optional fourth input argument of the solver. `options` is a natural name for the options structure, but you can use whatever name you like.

```
function ch2ex1
% x_1 = y(1), x_2 = y(2), y = y(3)

options = odeset('AbsTol',1e-20);
```

```
[t,y] = ode15s(@odes,[0 8e5],[0; 1; 0],options);
plot(t,y(:,1:2))
figure
semilogx(t,y(:,3))

%================================================
function dydt = odes(t,y)
k = [8.4303270e-10 2.9002673e+11 2.4603642e+10 8.7600580e-06];
dydt = zeros(3,1);
dydt(1) = -k(1)*y(1) + k(2)*y(3);
dydt(2) = -k(4)*y(2) + k(3)*y(3);
dydt(3) =  k(1)*y(1) + k(4)*y(2) - (k(2) + k(3))*y(3);
```

We solved this problem with the BDF code `ode15s` because the IVP is stiff. In simplest use, all the IVP solvers are interchangeable. To use the Runge–Kutta code `ode45` instead, all you must do is change the name of the solver in `ch2ex1.m`. You are asked to do this in Exercise 1.18 so that you can see for yourself why special methods are needed for stiff problems.

Example 1.3.3. By default the solvers return the solution at all steps in the course of the integration. To get more control over the output, you can write an *output function* that the solver will call at each step with the solution it has just computed. Several output functions are supplied with the solvers. As a convenience, if you do not specify any output arguments, the solver understands that you want to use the output function `odeplot`. This output function displays all of the solution components as they are computed. Exercises 1.15 and 1.18 are examples. Often you do not want to see all of the components. You can control which components are displayed with the `OutputSel` option. Other output functions make it convenient to plot in a phase plane. Exercise 1.21 has you experiment with one.

Along with the example programs of the text is a program `dfs.m` that provides a modest capability for computing and plotting solutions of a scalar ODE, $y' = f(t, y)$. Exercise **??** discusses how to use the program, but we consider it here only to show how to write an output function. Some solvers allow users to specify a minimum step size, *hmin*, and terminate the integration if a step size this small appears to be necessary. The MATLAB solvers do not provide for a minimum step size, but `dfs.m` shows how to obtain this capability with an output function. Relevant portions of the program are

```
function dfs(fun,window,npts)
.
.
.
hmin = 1e-4*(wR - wL);
.
.
.
options = odeset('Events',@events,'OutputFcn',@outfcn,...
.
.
.
[t,y] = ode45(@F,[t0,wR],y0,options);
plot(t,y,'r')
[t,y] = ode45(@F,[t0,wL],y0,options);
plot(t,y,'r')
.
.
.
function status = outfcn(t,y,flag)
```

```
global hmin
persistent previoust
status = 0;
switch flag
case 'init'
    previoust = t(1);
case ''
    h = t(end) - previoust;
    previoust = t(end);
    status = (abs(h) <= hmin);
case 'done'
    clear previoust
end
```

The option `OutputFcn` communicates to the solver the name of the output function, here `outfcn`. When there is an output function present, the solver calls it at each step with arguments `t,y,flag`. The arguments `t` and `y` are values of the independent and dependent variables and the string `flag` indicates the circumstances of the call. The output function returns a variable `status`. If you want to continue the integration, return a value of 0 (false), and if you want to terminate the run, return a value of 1 (true). The solver calls first with `flag` equal to `'init'` so that the output function can initialize itself. For example, if you wanted to write output to a file, you could open the file at this time. In this first call the solver provides the output function with the interval of integration $[a, b]$ in `t` and $y(a)$ in `y`. In the present example a variable `previoust` is initialized to the initial value of the independent variable, `t(1)`. It is declared to be `persistent` so that it will be retained between calls. After each step of the integration, the solver calls the output function with `flag` equal to `''`. Generally `t` is the value of the independent variable at the end of the step and `y` is the approximate solution there, but there is a complication in this example because the integration is done with `ode45`. The solvers have an option called `Refine` that can be given an integer value to have the solver return that many approximate solutions at equally spaced points in the span of the step. The default value of `Refine` is 1 for all the solvers except `ode45`, for which it is 4. You can do whatever you like with these approximate solutions in the output function. You might, for example, write `t` and selected components of `y` to a file. When `Refine` is larger than 1, the approximate solutions appear in order, so we can use `t(end)` to obtain the value of the independent variable at the end of the step that we need to compute the current step size in this example. If the step size is not larger than the minimum step size, `status` is set to 1 so that the integration will be terminated. Exercise 1.29 asks you to modify the output function of `dfs.m` so as to terminate the run in other circumstances, too. At the end of the run, the solver calls the output function with `flag` equal to `'done'` so that the output function can finish up. For example, it might close an output file. Here `previoust` is cleared just to illustrate the case.

Exercise 1.17. In Chapter **??** it was asserted that there is no essential difference solving numerically IVPs for the equations

- $y' = y^2 + 1$
- $y' = y^2 + t$
- $y' = y^2 + t^2$

See for yourself by solving the equations with, say, `ode45` on the interval $[0, 1]$ with initial value $y(0) = 0$ and plotting the solutions.

Exercise 1.18. Modify `ch2ex1.m` so that the approximate solutions are displayed as they are computed. As explained in Example 1.3.3, this is what happens when you do not specify any output arguments. You will find that `ode15s` solves the IVP easily. Although it appears that `ode45` is stuck at the initial point, it *is* solving the problem. This will be clear when you click the "Stop" button after you tire of waiting for the solver to finish up. At that time the numerical solution will

be plotted on a scale that shows how far the integration has progressed. You will find that `ode45` is solving the IVP, but it is advancing the integration *very* slowly.

Exercise 1.19. Before effective codes for stiff IVPs were widely available, some stiff problems were solved by singular perturbation methods, perhaps in combination with numerical methods for non–stiff IVPs. In this way Lapidus *et alia* [73] solve the following model of the thermal decomposition of ozone:

$$\begin{aligned} \frac{dx}{dt} &= -x - xy + \epsilon\kappa y \\ \epsilon\frac{dy}{dt} &= x - xy - \epsilon\kappa y \end{aligned}$$

Here x is the reduced ozone concentration, y is the reduced oxygen concentration, and the parameters $\epsilon = 1/98$ and $\kappa = 3$. (Notice that $y'(t)$ is multiplied by the small parameter ϵ.) The IVP is to be solved on $[0, 240]$ with initial conditions $x(0) = 1, y(0) = 0$. Lapidus *et alia* observe that singular perturbation methods are not very accurate for this IVP. Ironically, this is because ϵ is not very small, which is to say that the IVP is not very stiff. Nowadays numerical solution of this IVP is routine. Solve it with `ode15s` and plot $y(t)$ with `semilogx` and `axis([0.01 100 0 1])` for comparison with Fig. 3 of [73]. Verify that the IVP is not very stiff by showing that you can solve the IVP easily with `ode45`. Using `odeset`, set the option `Stats` to `on` for the two runs to compute statistics that will help with this verification.

Exercise 1.20. The text states that when evaluating an implicit method, the iteration matrix is ill–conditioned if the IVP is stiff. See for yourself by modifying `ch2ex1.m` so as to display a condition number for each of the iteration matrices formed in the course of the integration. For this you will need to copy `ode15s.m` to your working directory and rename it to, say, `mode15s.m`. The source code for `ode15s` is found in the subdirectory `/toolbox/matlab/funfun/` of your installation directory for MATLAB. You will also need copies of some auxiliary functions that `ode15s` calls, namely `ntrp15s.m`, `odearguments.m`, `odeevents.m`, `odejacobian.m`, and `odemass.m`. They are found in `/toolbox/matlab/funfun/private/`. To estimate a condition number for the iteration matrix and communicate it to the main program, search `mode15s.m` for the command

```
[L,U] = lu(Miter);
```

which uses the MATLAB function `lu` to compute an LU factorization. It occurs twice, once as the solver initializes and once in the main loop. The first time you might follow this command with

```
global tcond cond
tcond = t;
cond = condest(Miter);
```

This initializes a pair of output arrays that keep track of where the iteration matrix is formed, computes an estimate of its condition in the 1–norm using the MATLAB function `condest`, and makes this information available outside the solver. You might then follow the second appearance of the command by

```
tcond = [tcond t];
cond = [cond condest(Miter)];
```

to extend the output arrays each time an iteration matrix is formed and factored. After this preparation it is easy to monitor a condition number for the iteration matrix as you solve an IVP with `mode15s`. To do this with `ch2ex1.m`, all you must do is add

```
global tcond cond
```

to gain access to the information and follow the computations with a plot such as

```
figure
loglog(tcond,cond,'*-')
```

You will find that in the first part of the integration, the condition number is comparable to 1, the smallest possible value. It then grows steadily to values comparable to $\frac{1}{\texttt{eps}}$, the largest possible value. The condition number is reduced somewhat for the last step. Using the appearance of the solution components as plotted by `ch2ex1.m` and a plot of the step sizes,

```
figure
loglog(t(1:end-1),diff(t))
```

explain why you might have expected the condition number to behave as described.

Exercise 1.21. Two of the output functions that accompany the MATLAB IVP solvers, `odephas2` and `odephas3`, plot solutions in a phase space as they are computed. Solutions of the ODEs

$$\begin{aligned} y_1' &= -y_2 - \frac{y_1 y_3}{r} \\ y_2' &= y_1 - \frac{y_2 y_3}{r} \\ y_3' &= \frac{y_1}{r} \end{aligned}$$

where

$$r = \sqrt{y_1^2 + y_2^2}$$

lie on a torus in phase space. Using `ode45`, solve these equations on the interval $0 \le t \le 10$ with initial values

$$(y_1(0), y_2(0), y_3(0)) = (3, 0, 0)$$

and plot the solution in three–dimensional phase space as it is computed by setting `OutputFcn` to `@odephas3`. The plot routines draw straight lines between successive output points. When the step sizes are "large", the output points may be so far apart that the graph is not smooth. This is more likely to happen when plotting in phase space. If it happens, you can use the option `Refine` to have the solver compute additional output points in the span of each step by evaluating a continuous extension. To see this effect clearly, compute two figures. For one figure set `Refine` to 1 so that there is one output point per step. For `ode45` the default value of `Refine` is 4, but this graph would benefit from more output points, so for the second figure, set `Refine` to 10. The auxiliary function `odeset` allows you to alter options as well as set them: If you compute the first figure with

```
options = odeset('OutputFcn',@odephas3,'Refine',1);
```

you can reset the value of `Refine` for the second figure with

```
options = odeset(options,'Refine',10);
```

Exercise 1.22. Raghothama and Narayanan [103] consider the effects of parametric excitation for the ODE

$$x''(t) = -2\zeta x'(t) - x - 1.5x^2(t) - 0.5x^3(t) + f_2 \cos(\Omega_2 t) + f_3 \cos(\Omega_3 t)$$

where $f_2 = 0.05$, $\Omega_2 = 1$, $\Omega_3 = 2$, $\zeta = 0.1$, and f_3 is treated as a bifurcation parameter. The authors demonstrate that as f_3 is changed, the phase curves $(x(t), x'(t))$ can be qualitatively different. Use `ode45` to solve the ODE for the following four cases corresponding to the cases in Fig. 9 of [103]. Integrate with `RelTol = 1e-8` and `AbsTol = 1e-8` from $t = 0$ to $t = 500$. To show the limiting behavior, just plot the phase curve for $400 \le t \le 500$. If you have the integration coded as `[t,x] = ode45...`, you can do this by first finding the indices of mesh points in this interval with `ndx = find(t >= 400)` and then plotting with `plot(x(ndx,1),x(ndx,2))`. Your curves should represent periodic motion of period 1, 2, 4, and 8, respectively.

- $x(0) = 0$, $x'(0) = -0.60$, $f_3 = 0.50$
- $x(0) = 0$, $x'(0) = -0.80$, $f_3 = 0.92$
- $x(0) = 0$, $x'(0) = -0.59$, $f_3 = 0.99$

- $x(0) = 0$, $x'(0) = -0.80$, $f_3 = 1.005$

The authors further demonstrate that chaotic behavior can occur. Solve the IVP in the same way with $x(0) = 0$, $x'(0) = 0$, and $f_3 = 1.35$. Your phase curve should resemble somewhat a right–handed catcher's mitt.

1.3.1 Event Location

Along with the example programs of the text is a program `dfs.m` that provides a modest capability for computing and plotting solutions of a scalar ODE, $y' = f(t, y)$. Exercise **??** discusses how to use this program, but only a few details are needed here. Solutions are plotted in a window that you specify by an array $[wL, wR, wB, wT]$. When you click on a point (t_0, y_0), the program computes the solution $y(t)$ that has the initial value $y(t_0) = y_0$ and plots it as long as $wL \le t \le wR$, $wB \le y \le wT$. By calling `ode45` with interval $[t_0, wR]$, the program computes $y(t)$ from the initial point to the right edge of the plot window. It then calls the solver with the interval $[t_0, wL]$ to compute $y(t)$ from the initial point to the left edge. But what if the solution goes out the bottom or top of the plot window along the way? What we need is the capability of terminating the integration if there is a t^* where either $y(t^*) = wB$ or $y(t^*) = wT$.

While approximating the solution of an IVP, we sometimes need to locate points t^* where certain *event functions*

$$g_1(t, y(t)), g_2(t, y(t)), \ldots, g_k(t, y(t))$$

vanish. Determining these points is called *event location*. Sometimes we just want to know the solution, $y(t^*)$, at the time, t^*, of the event. On other occasions we need to terminate the integration at t^* and possibly start solving a new IVP with initial values and ODEs that depend on t^* and $y(t^*)$. Sometimes it matters whether an event function is decreasing or increasing at the time of an event. In the `dfs.m` program, there are two event functions, $wB - y$ and $wT - y$, both events are terminal, and we don't care whether the event function is increasing or decreasing at an event. All the MATLAB IVP solvers have a powerful event location capability. Most continuous simulation languages and a few of the IVP solvers widely used in general scientific computing have some kind of capability of locating events. Event location can be ill–posed and some of the algorithms in use are crude. The difficulties of this task are often not appreciated, so one of the aims of this section is to instill some caution when using an event location capability. A deeper discussion of the issues can be found in [114, 118].

Most codes that locate events monitor the event functions for a change of sign in the span of a step. If, say, $g_i(t_n, y_n)$ and $g_i(t_{n+1}, y_{n+1})$ have opposite signs, the solver uses standard numerical methods to find a root of the algebraic equation $g_i(t, y(t)) = 0$ in $[t_n, t_{n+1}]$. In finding a root, it is necessary to evaluate the event function g_i at a number of points t. Here is where a continuous extension is all but indispensable. Without a continuous extension, for each value of t in the interval $[t_n, t_{n+1}]$ where we must evaluate g_i, we must take a step with the ODE solver from t_n to t to compute the approximation to $y(t)$ that we need in evaluating $g_i(t, y(t))$. It is much more efficient simply to evaluate a polynomial approximation $S(t)$ to $y(t)$ that is accurate for all of the interval $[t_n, t_{n+1}]$. Locating the event then amounts to computing zeros of the equation $g_i(t, S(t)) = 0$. Often there are a number of event functions. They must be processed simultaneously because it is possible that in the course of locating a root of one event function, we discover that another event function has a root and it is closer to t_n. We must find the event closest to t_n because the definition of the ODE might change there.

This outline of event location gives us good reasons for caution. The approach has no hope of finding events characterized by an even order zero because there is no change of sign. It is entirely possible to miss a change of sign at an odd order zero. That is because the step size is chosen to resolve changes in the solution, not changes in the event functions, so the code could step over several events and have no sign change at the end of the step. A related difficulty is that in applications, we want the first root, the one closest to t_n, and in general there is no way to be sure we compute it. Some of the event functions that arise in practice are not smooth, which makes the computation of roots more difficult. Indeed, discontinuous event functions are not rare. As when computing

roots of any algebraic equation, the root of an event function can be ill–conditioned, meaning that the value t^* is poorly determined by the values of the event function. Generally in root–solving we assume that the function can be evaluated very accurately, but the situation is different here because we only compute the argument $y(t)$ to a specified accuracy. A fundamental issue, then, is how well t^* is determined when we compute the function $y(t)$ to a specified accuracy. That must be considered when formulating the problem, but it has a consequence for the root–solver, namely, how accurately should we have it compute t^*? Some codes ask users to supply tolerances for event location in addition to those for the integration. It is difficult to choose sensible values, so other codes, including those of MATLAB, take a different approach. They simply locate events about as accurately as possible in the precision of the computer. For this approach to be practical, the algorithm for computing roots must be fast in the usual case of smooth event functions. It must also be reasonably fast when an event function is not smooth. An article by Moler [87] explains how the algorithm of MATLAB combines the robust and reasonably fast method of bisection with a scheme that is both fast for smooth event functions and vectorizable.

The examples that follow show that it is not hard to do event location with the MATLAB IVP solvers. Indeed, the only complication in event location itself is that you must specify what kinds of events you want to locate and what you want the solver to do when it finds one. On the other hand, problems involving event location are often rather complicated because of what is done after an event occurs. For Example 1.3.4 this is just a matter of processing the events. Exercises 1.25 and 1.26 take up problems of this kind. Example 1.3.5 is complicated because a new IVP is formulated and solved after each event. Exercises 1.27 and 1.28 take up variants of this example. An example and a couple of exercises in later sections illustrate event location when there is something new about solving the ODEs: The ODEs of Example 1.3.7 are formulated in terms of a mass matrix. Exercise 1.32 is an extension of this example. The ODEs of Exercise 1.37 are singular at the initial point.

To see that event location does not need to be complicated, let us look at the relevant portions of `dfs.m`. There is a function `events` that evaluates the event functions and tells the solver what it is to do. A handle for this function is provided as the value of the option `Events`. The values of the two event functions for the input values `t,y` are returned by `events` in the output argument `value`. The vector `isterminal` tells the solver whether the events are terminal. Here both entries are ones, meaning that both are terminal events. The vector `direction` tells the solver if it matters whether the event function is increasing or decreasing at an event. Here both entries are zeros, meaning that the direction does not matter. The solver has some extra output arguments when there are events, but here all we want to do is terminate the integration and so do not ask for this optional output.

```
function dfs(fun,window,npts)
.
.
.
options = odeset('Events',@events,...
.
.
.
[t,y] = ode45(@F,[t0,wR],y0,options);
plot(t,y,'r')
[t,y] = ode45(@F,[t0,wL],y0,options);
plot(t,y,'r')
.
.
.
function [value,isterminal,direction] = events(t,y)
global wB wT
value = [wB; wT] - y;
isterminal = [1; 1];
direction = [0; 0];
```

It is nearly as easy to solve the event location problem of Exercise 1.23, but it is convenient then to use the solver's optional output arguments. This is just a sketch of event location; details are found in the examples that follow.

Example 1.3.4. Poincaré maps are important tools for the interpretation of dynamical systems. Very often they are plots of the values of the solution at a sequence of equally spaced times. These values are easily obtained by simply specifying an array `tspan` of these times. However, sometimes values are sought for which a linear combination of the solution values vanishes, and event location is needed to obtain these values. Koçak does this with the `PHASER` program in Lessons 13 and 14 of [70]. In his example four ODEs for the variables $x_1(t)$, $x_2(t)$, $x_3(t)$, and $x_4(t)$ describing a pair of harmonic oscillators are integrated. The simple equations and initial conditions are found in the function `ch2ex2.m` below. Plotting $x_1(t)$, $x_2(t)$, and $x_3(t)$ shows what appears to be a finite cylinder. As coded, you can click on the `Rotate 3D` tool and drag the figure to come to a good understanding of this projection of the solution. One view is provided here as Fig. 1.2. The `PHASER` program can find and plot points where

$$Ax_1(t) + Bx_2(t) + Cx_3(t) + D = 0$$

Koçak computes two such Poincaré maps. One is a plot of the coordinates $(x_1(t^*), x_3(t^*))$ for values of t^* such that $x_2(t^*) = 0$. The other is a plot of the coordinates $(x_1(t^*), x_2(t^*))$ for values of t^* such that $x_3(t^*) = 0$. We accomplish this by employing two event functions, $g_1 = x_2$ and $g_2 = x_3$, that we evaluate in a subfunction called here `events`. As with other optional input, you tell the solver about this function by setting the option `Events` to the handle of this function. The event function must have as its input the arguments `t,x` and return as output a column vector `value` of the values of the event functions $g_1(t, x)$ and $g_2(t, x)$. We must write this function so that it also returns some information about what we want the code to do. For some problems we will want to terminate the integration when a certain event occurs. The second output argument of the event function is a column vector `isterminal` of zeros and ones. If we want to terminate the integration when $g_k(t, x) = 0$, we set `isterminal(k)` to 1 and otherwise we set it to 0. The initial point is a special case in this respect because sometimes an event that we want to regard as terminal occurs at the initial point. This special situation is illustrated by Example 1.3.5 and Exercise 1.24. Because this situation is not unusual, the solvers treat any event at the initial point as *not* terminal. Sometimes it matters whether the event function is decreasing or increasing at an event. The third output argument of the event function is a column vector `direction`. If we are interested in events determined by the equation $g_k(t, x) = 0$ only at locations where the event function g_k is decreasing, we set `direction(k)` to -1. If we are interested in events only when the function g_k is increasing, we set it to $+1$. If we are interested in all events for this function, we set `direction(k)` to 0. For the current example, events are not terminal and we want to compute all events.

When we use the event location capability, the solver returns some additional quantities as exemplified here by `[t,x,te,xe,ie] = ode45...`. The array `te` contains the values of the independent variable where events occur and the array `xe` contains the solution there. The array `ie` contains an integer indicating which event function vanished at the corresponding entry of `te`. (If you prefer to have the output in the form of a solution structure `sol`, this information about events is returned in the fields `sol.xe, sol.ye, sol.ie`, respectively.) If there are no events, all these arrays are empty, so a convenient way to check this possibility is to test `isempty(ie)`. One of the complications of event location is that if there is more than one event function, we do not know in what order the various kinds of events will occur. This complication is resolved easily with the `find` function as illustrated by this example. For instance, the command

```
event1 = find(ie == 1);
```

returns in `event1` the indices that correspond to the first event function. They are then used to extract the corresponding solution values from `xe` for plotting. All the MATLAB IVP solvers have event location and they are all used in exactly the same way. Here we use `ode45` and we use tolerances more stringent than the default values because we require an accurate solution over a good many periods. Fig. 1.3 shows one of the two Poincaré maps computed by this program. The program also reports that

```
Event 1 occurred 43 times.
Event 2 occurred 65 times.


function ch2ex2
opts = odeset('Events',@events,'RelTol',1e-6,'AbsTol',1e-10);
[t,x,te,xe,ie] = ode45(@odes,[0 65],[5; 5; 5; 5],opts);
plot3(x(:,1),x(:,2),x(:,3));
xlabel('x_1(t)'), ylabel('x_2(t)'), zlabel('x_3(t)')
if isempty(ie)
    fprintf('There were no events.\n');
else
    event1 = find(ie == 1);
    if isempty(event1)
        fprintf('Event 1 did not occur.\n');
    else
        fprintf('Event 1 occurred %i times.\n',length(event1));
        figure
        plot(xe(event1,1),xe(event1,3),'*');
    end
    event2 = find(ie == 2);
    if isempty(event2)
        fprintf('Event 2 did not occur.\n');
    else
        fprintf('Event 2 occurred %i times.\n',length(event2));
        figure
        plot(xe(event2,1),xe(event2,2),'*');
    end
end

%=============================================================
function dxdt = odes(t,x)
a = 3.12121212;
b = 2.11111111;
dxdt = [a*x(3); b*x(4); -a*x(1); -b*x(2)];

function [value,isterminal,direction] = events(t,x)
value = [x(2); x(3)];
isterminal = [0; 0];
direction = [0; 0];
```

Example 1.3.5. An interesting problem that is representative of many event location problems models a ball bouncing down a ramp as in Fig. 1.4. For simplicity we take the ramp to be the long side of the triangle with vertices $(0,0)$, $(0,1)$, and $(1,0)$. At time t the ball is located at the point $(x(t), y(t))$. We'll suppose that it is released from rest at a point above the end of the ramp, i.e., we start from $x(0) = 0$, $y(0) > 1$, $x'(0) = 0$, and $y'(0) = 0$. In free fall the motion of the ball is described by the equations

$$x'' = 0, \quad y'' = -g$$

where the second equation represents the acceleration due to gravity and the gravitational constant is taken to be $g = 9.81$. We must write these equations as a system of first order equations, which we do by introducing variables

$$y_1(t) = x(t),\ y_2(t) = x'(t),\ y_3(t) = y(t),\ y_4(t) = y'(t)$$

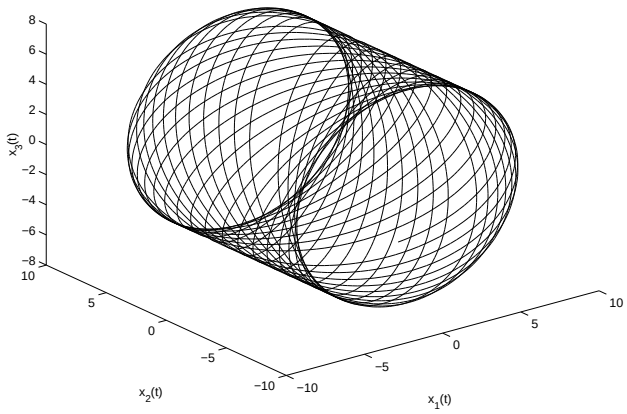

Figure 1.2: Lissajous figure for a pair of harmonic oscillators.

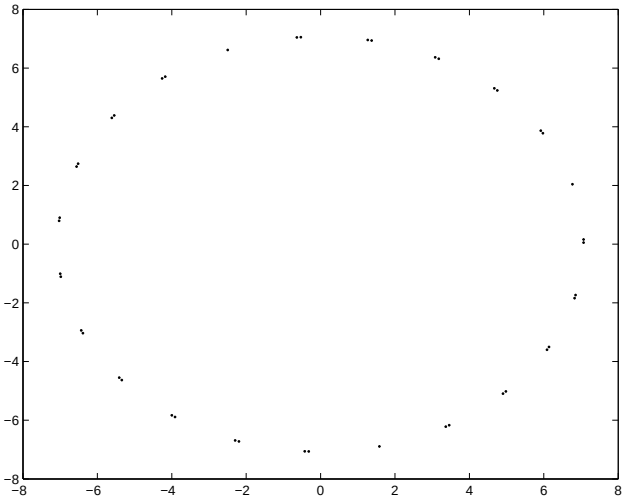

Figure 1.3: Poincaré map for a pair of harmonic oscillators.

These equations hold until the ball hits the ramp at a time $t^* > 0$. At this time the ball has moved to the right a distance $x(t^*)$ where the ramp has height $1 - x(t^*)$. (The first event has $x(t^*) = 0$.) The height of the ball is $y(t)$, so the event that the ball hits the ramp is

$$y(t^*) = 1 - x(t^*)$$

From this we see that the event function is

$$g_1 = y_3 - (1 - y_1)$$

This event is terminal because the model changes there. It is essential to locate the event in order to correctly model the subsequent trajectory of the ball since there is nothing in the equations themselves to change the direction of the ball if a bounce time event is missed by the code.

After hitting the ramp, the ball rebounds and its subsequent motion is described by another IVP for the same ODEs. The initial position of the ball in the new integration is the same as the terminal position in the old. The effect of a bounce appears in the initial velocity of the new integration which has a direction related to the velocity at t^* by the geometry of the ramp and a magnitude reduced by a coefficient of restitution k, a constant with value $0 < k < 1$. We'll not go into the details, but

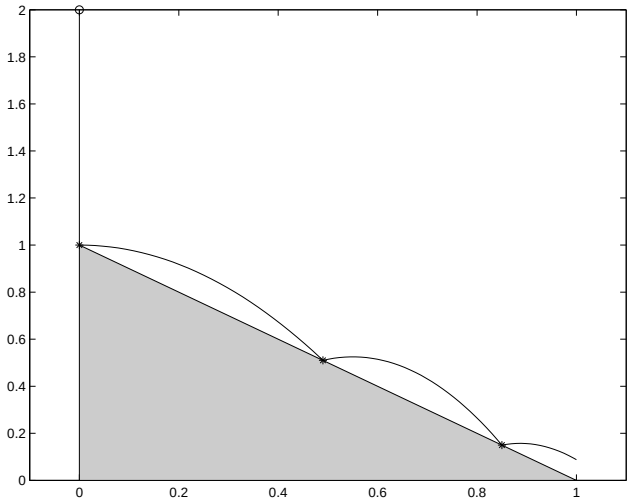

Figure 1.4: Path of the ball when the coefficient of restitution is $k = 0.35$.

the result is that the initial conditions for the new integration starting at time at t^* are

$$(y_1(t^*),\, -ky_4(t^*),\, y_3(t^*),\, ky_2(t^*))$$

The same terminal event function is used to recognize the next time that the ball hits the ramp. However, at the initial point of this new IVP the ball is touching the ramp, so the terminal event function vanishes there. This shows why we want to treat the initial point as a special case for terminal events. There is another way to deal with this kind of difficulty that is convenient here. If we set `direction` to 0, the solvers report an event at the initial point in addition to the one that terminates the integration. We can avoid this by setting `direction` to -1. The distance of the ball from the ramp is increasing at the initial point, so with this setting, the solver ignores the zero there and reports only the event of the ball dropping to the ramp. We repeat this computation of the path of the ball from bounce to bounce as it goes down the ramp.

The computation is complete when the ball reaches the end of the ramp which is at $x(t^*) = 1$. To recognize this, we use the terminal event function $g_2 = y_1 - 1$. For some values of k and some choices of initial height of the ball, the bounces cluster and the ball never reaches the end of the ramp. Our equations cannot model what actually happens then—the ball rolls down the ramp after it has finished bouncing. So, we must recognize that the bounces are clustering both to avoid clutter in the plot and stop the integration. In the program this is done by quitting when the times at which successive bounces occur differ by less than 1%. There is a related difficulty when the ball bounces very nearly at the end of the ramp, so we also quit if it bounces at a point that is less than 1% of the distance from the end.

Some thought about the problem or some experimentation brings to our attention a difficulty of some generality, namely that we do not know how long it will take for the ball to reach the end of the ramp, if ever. Certainly the terminal time depends on the initial height and on the coefficient of restitution, k. The solvers require us to specify an interval of integration, so we must guess a final time. If this guess isn't large enough, we'll need to continue integrating. Some IVP codes have a capability for *extending the interval*, but the MATLAB solvers do not. Instead, we solve a new IVP with initial data taken from the final data of the previous integration. This is an acceptable way to proceed because the MATLAB solvers start themselves efficiently and it won't be necessary to restart many times if our guesses are at all reasonable. Exercises 1.27 and 1.28 have you solve variants of the bouncing ball problem.

A couple of programming matters deserve comment. The path of the ball consists of several pieces, either because of bounces or because we had to extend the interval of integration. It is convenient to accumulate the whole path in a couple of arrays in a manner illustrated by the

program. Because the ODEs are so easy to integrate, the solver does not take enough steps to result in a smooth graph. An array `tspan` is used to deal with this. In addition to plotting the solution with a smooth curve, we mark the initial point with an 'o' and the points of contact with the ramp with an '*'.

```
function ch2ex3
% The ball is at (x(t),y(t)).
% Here y(1) = x(t), y(2) = x'(t), y(3) = y(t), y(4) = y'(t).

% Set initial height and coefficient of restitution:
y0 = [0; 0; 2; 0];
k = 0.35;

% Plot the initial configuration:
fill([0 0 1],[0 1 0],[0.8 0.8 0.8]);
axis([-0.1 1.1 0 y0(3)])
hold on
plot(0,y0(3),'ro'); % Mark the initial point.

options = odeset('Events',@events);
% Accumulate the path of the ball in xplot,yplot.
xplot = [];
yplot = [];
tstar = 0;
while 1
    tspan = linspace(tstar,tstar+1,20);
    [t,y,te,ye,ie] = ode23(@odes,tspan,y0,options);
    % Accumulate the path.
    xplot = [xplot; y(:,1)];
    yplot = [yplot; y(:,3)];
    if isempty(ie)          % Extend the interval.
        tstar = t(end);
        y0 = y(end,:);
    elseif ie(end) == 1     % Ball bounced.
        plot(ye(end,1),ye(end,3),'r*'); % Mark the bounce point.
        if (te(end) - tstar) < 0.01*tstar
            fprintf('Bounces accumulated at x = %g.\n',ye(end,1))
            break;
        end
        if abs(ye(end,1) - 1) < 0.01
            break;
        end
        tstar = te(end);
        y0 = [ye(end,1); -k*ye(end,4); ye(end,3); k*ye(end,2)];
    elseif ie(end) == 2     % Reached end of ramp.
        break;
    end
end
plot(xplot,yplot);

%================================================
function dydt = odes(t,y)
dydt = [y(2); 0; y(4); - 9.81];

function [value,isterminal,direction] = events(t,y)
```

```
value = [ y(3) - (1 - y(1))
             y(1) - 1      ];
isterminal = [1; 1];
direction  = [-1; 0];
```

Exercise 1.23. J.R. Dormand [34, p. 114] sets an amusing exercise about the ejection of a cork from a bottle containing a fermenting liquid. Let $x(t)$ be the displacement of the cork at time t and let L be the length of the cork. While $x(t) \le L$ (the cork is still in the neck of the bottle), the displacement satisfies

$$\frac{d^2x}{dt^2} = g(1+q)\left[\left(1+\frac{x}{d}\right)^{-\gamma} + \frac{Rt}{100} - 1 + \frac{qx}{L(1+q)}\right]$$

Here the physical parameters are $g = 9.81$, $q = 20$, $d = 5$, $\gamma = 1.4$, and $R = 4$. If $x(0) = 0$ and $x'(0) = 0$ (that is, the cork starts at rest) and $L = 3.75$, find when $x(t^*) = L$ (that is, the cork leaves the neck of the bottle). Also, what is $x'(t^*)$ (the speed of the cork as it leaves the bottle)? This is a simple event location problem. Integrate the IVP until the terminal event $x(t^*) - L = 0$ occurs and then print out t^* and $x'(t^*)$. The only difficulty is that the MATLAB solvers require you to specify an interval of integration. A simple way to proceed is to guess that the cork will be ejected by, say, $t = 100$ and ask the code to integrate over $[0, 100]$. You then need to check that the integration was actually terminated by the event. You can do this by testing whether the integration reached the end of the interval or by testing whether the array `ie` is empty. If there was no event, you'll need to try again with a longer interval.

Exercise 1.24. In Chapter **??** the planar motion of a shot fired from a cannon is discussed. This motion is governed by the ODEs

$$\begin{aligned} y' &= \tan(\phi) \\ v' &= -\frac{g\sin(\phi) + \nu v^2}{v\cos(\phi)} \\ \phi' &= -\frac{g}{v^2} \end{aligned}$$

where y is the height of the shot above the level of the cannon, v is the velocity of the shot, and ϕ is the angle (in radians) of the trajectory of the shot with the horizontal. The independent variable x measures the distance from the cannon. The constant ν represents air resistance (friction) and $g = 0.032$ is the appropriately scaled gravitational constant. A natural question is to determine the range of a shot for given muzzle velocity $v(0)$ and initial angle of elevation $\phi(0)$. The initial height $y(0) = 0$, so we want the first $x^* > 0$ for which $y(x)$ vanishes. This is an event location problem for which a terminal event occurs at the initial point. Chapter **??** discusses another natural question that leads to a BVP: For what initial angle $\phi(0)$ does the cannon have a given range? Fig. **??** shows two trajectories for muzzle velocity $v(0) = 0.5$ that have range 5 when $\nu = 0.02$. In computing this figure it was found that the two trajectories have $\phi(0) \approx 0.3782$ and $\phi(0) \approx 9.7456$. For each of these initial angles, solve the IVP with the given values of $v(0)$ and ν and use event location to verify that the range is approximately 5.

Exercise 1.25. Integrate the pendulum equation

$$\theta'' + \sin(\theta) = 0$$

with initial conditions $\theta(0) = 0$ and $\theta'(0) = 1$ on the interval $[0, 20\pi]$. The solution oscillates a number of times in this interval, so to be more confident of computing an accurate solution, set `RelTol` to 10^{-5} and `AbsTol` to 10^{-10}. Locate all the points t^* where $\theta(t^*) = 0$. Plot $\theta(t)$ and the locations of these events. Compute and display the spacing between successive zeros of $\theta(t)$. In MATLAB this can be done conveniently by computing the spacing with `diff(te)`. The solution is periodic, so if it has been computed accurately, this spacing will be nearly constant. For solutions $\theta(t)$ that are "small", the ODE is often approximated by the linear ODE $x'' + x = 0$. The spacing

between successive zeros of $x(t)$ is π. How does this compare to the spacing you found for the zeros of $\theta(t)$? Reduce the size of $\theta(t)$ by reducing the initial slope to $\theta'(0) = 0.1$. Is the spacing between zeros then closer to π? What if you reduce the initial slope to $\theta'(0) = 0.01$?

Exercise 1.26. When solving IVPs typically we make up a table of the solution for given values of the independent variable. However, sometimes we want a table for given values of one of the dependent variables. This can be done using event location. An example is provided by Moler and Solomon [88] who consider how to compute a table of values of the integral

$$t(x) = \int_{x_0}^{x} \frac{1}{\sqrt{f(s)}}\, ds$$

for a smooth function $f(x) > 0$. Computing the integral is equivalent to solving the IVP

$$\frac{dt}{dx} = \frac{1}{\sqrt{f(x)}}, \qquad t(x_0) = 0$$

This ODE will be difficult to solve as x approaches a point where $f(x)$ vanishes. To deal with this, Moler and Solomon first note that $x(t)$ satisfies the IVP

$$\frac{dx}{dt} = \sqrt{f(x)}, \qquad x(0) = x_0$$

Because of the square root, this equation also presents difficulties as $x(t)$ approaches a point where $f(x(t))$ has a change of sign, or equivalently, where $x'(t)$ vanishes, so they differentiate it to obtain the IVP

$$x'' = 0.5f'(x), \quad x(0) = x_0, \quad x'(0) = \sqrt{f(x_0)}$$

There is no difficulty integrating this equation through a point where $x'(t)$ vanishes, so such a point can be located easily as a terminal event. We want the values t_i where $x(t)$ has given values x_i, i.e., we want to locate where the functions $x(t) - x_i$ vanish, a collection of non–terminal events. Write a program to make up a table of values of the integral for given x_i. You could hard code the values x_i in your event function, but a more flexible approach is to define an array `xvalues` of the x_i in the main program and pass it as a `global` variable to your event function. Check out your program by finding values of the integral for $x_0 = 0$ and `xvalues = 0.1:0.1:0.9` when $f(x) = x$. In this use `ode45` and integrate over $0 \le t \le 2$. Verify analytically that the integral is $t(x) = 2\sqrt{x}$ and use this to compare the values you compute to the true values. The fact that $f(x)$ vanishes at the initial point does not complicate the numerical integration, but it does mean that there is a terminal event at the initial point. Recall that this is a special case for event location. The solvers report such an event, but do not terminate the integration. After checking out your program in this way, modify it to make up a table of values of the integral when $f(x) = 1 - x$ and $f(x) = 1 - x^2$.

Exercise 1.27. The program `ch2ex3.m` plots the path of a ball bouncing down a ramp when the coefficient of restitution $k = 0.35$. Experiment with the effects of changing k. In particular, reduce k sufficiently to see an example of bounces clustering. You might, for example, try $k = 0.6$ and $k = 0.3$.

Exercise 1.28. Suppose that in the configuration of Example 1.3.5, there is a vertical wall located at the end of the ramp, i.e., where $x(t) = 1$. Modify `ch2ex3.m` so as to compute the path of the ball then. An easy way to show the wall is to make it the right side of the plot by changing the `axis` command to `axis([-0.1 1 0 y0(3)])`. Without the wall, the computation of the path is terminated when the ball reaches the end of the ramp as reported by `ie(end) == 2`. With a wall at the end of the ramp, this event corresponds to hitting the wall. After hitting the wall, continue integrating with initial values

$$(y_1(t^*),\ -ky_2(t^*),\ y_3(t^*),\ y_4(t^*))$$

If the ball is nearly in the corner, say $y_3(t^*) \le 0.01$, terminate the computation. Find the path of the ball for $k = 0.35$ and $k = 0.7$.

Exercise 1.29. It is natural to use event location in `dfs.m` to terminate the integration when $y(t)$ reaches the bottom or the top of the plot window, but that is not the only way to do this. The plot routine displays only the portion of $y(t)$ that lies within the plot window, so all we need to do is stop integrating when the solver steps outside the window. This can be done within the output function. Make yourself a copy of `dfs.m` and give it a different name. Remove the event function from this program and modify the output function so that the run is terminated if either `y(end) < wB` or `y(end) > wT`. This is cheaper than event location because it avoids the expense of locating accurately the time t^* where $y(t)$ reaches either the bottom or top of the plot window.

1.3.2 ODEs Involving a Mass Matrix

Some solvers accept systems of ODEs written in forms more general than $y' = f(t, y)$. In particular, the MATLAB IVP solvers accept problems of the form

$$M(t, y)y' = f(t, y) \tag{1.37}$$

involving a mass matrix $M(t, y)$. If $M(t, y)$ is singular, this is a system of *differential algebraic equations* (DAEs). They are closely related to ODEs, but differ in important ways. We do not pursue the solution of DAEs in this book beyond noting that the MATLAB codes for stiff ODEs can solve DAEs of index 1 arising in this way [117]. In §1.3.3 we discuss an approach to solving PDEs numerically that leads to systems of ODEs with a great many unknowns. Some of the schemes result in equations of the form (1.37). Example 1.3.11 provides more details about solving such problems.

In simplest use, the only difference when solving a problem of the form (1.37) with one of the MATLAB IVP solvers instead of a problem in standard form is that you must inform the solver about the presence of $M(t, y)$ by means of the option `Mass`. If the matrix is a constant, you should provide the matrix itself as the value of this option. The codes exploit constant mass matrices, so this is efficient as well as convenient. Suppose, for example, that you choose an explicit Runge–Kutta method. You provide the ODEs in the convenient form (1.37), but the solver actually applies the Runge–Kutta method to the ODEs

$$y' = F(t, y) \equiv M^{-1} f(t, y)$$

which is in standard form. As the code initializes itself, it computes an LU factorization of M. Whenever it needs to evaluate $F(t, y)$, it evaluates $f(t, y)$ and solves a linear system for F using the stored factorization of M. For the solvers that proceed in this way, allowing a mass matrix is mainly a convenience for you. However, other solvers modify the methods themselves to include a mass matrix. Without going into the details, it should be no surprise that the iteration matrix for the BDFs is changed from $I - h\gamma J$ to $M - h\gamma J$. The iteration matrix must be factored in any case, so the extra cost of the more general form is unimportant. When the mass matrix is not constant, the modifications to the methods are more substantial. If this case, you must provide a (sub)function for evaluating the mass matrix and pass the name to the solver as the value of `Mass`.

To solve problems of the form (1.37), all that you *must* do is provide the mass matrix necessary to define the ODEs. The solver needs to know whether the mass matrix is singular at the initial point, that is, whether the problem is a DAE. The option `MassSingular` has values `yes`, `no`, and the default of `maybe`. The default causes the solver to perform a numerical test for singularity that you can avoid if you know whether you are solving ODEs or DAEs. For large systems it is essential to take advantage of sparsity and inform the solver about how strongly the mass matrix $M(t, y)$ depends on y, matters discussed in Example 1.3.11.

Example 1.3.6. MATLAB 6 comes with a demonstration code, `batonode`, illustrating the use of a mass matrix. It is an interesting problem that is not described in the documentation, so we explain it here. It is based on Example 4.3A of [130]. A baton is modeled as two particles of masses m_1 and m_2, respectively, fastened to opposite ends of a light straight rod of length L. The motion of the baton is followed in a vertical plane under the action of gravity. If the coordinates of the first particle at time t are $(X(t), Y(t))$ and the angle the rod makes with the horizontal is $\theta(t)$,

Lagrange's equations for the motion are naturally expressed in terms of a mass matrix that involves the unknown $\theta(t)$. The ODEs are written in terms of the vector

$$y = (X,\, X',\, Y,\, Y',\, \theta,\, \theta')^T$$

and the functions $f(t, y)$ and $M(t, y)$ are defined by the partial listing of `batonode` that follows. (We do not display the comments nor the coding associated with plotting the motion of the baton.) Clearly, it is convenient to solve this small, non–stiff problem using the natural formulation in terms of a mass matrix rather than convert it to the usual standard form. This program exploits the capability of *passing parameter values* through the solver to the functions defining the ODEs and the mass matrix as arguments at the ends of the lists of input variables. This capability is common among the functions of MATLAB, so we content ourselves with an example of its use. The option `MassSingular` could have been set to `no`, but the default test for this is inexpensive with only 6 equations. Also, there is no advantage to be gained from sparsity when there are only 6 equations, but to exploit sparsity in this program, all you must do is change `M = zeros(6,6)` to `M = sparse(6,6)`. Fig. 1.5 is the result of running the code `batonode` with the title removed from the figure. (Exercise 1.31 has you experiment further with `batonode`.)

```
function batonode
      .
      .
      .
m1 = 0.1;
m2 = 0.1;
L = 1;
g = 9.81;

tspan = linspace(0,4,25);
y0 = [0; 4; 2; 20; -pi/2; 2];

options = odeset('Mass',@mass);
[t y] = ode45(@f,tspan,y0,options,m1,m2,L,g);
      .
      .
      .
% --------------------------------------------

function dydt = f(t,y,m1,m2,L,g)
dydt = [
 y(2)
 m2*L*y(6)^2*cos(y(5))
 y(4)
 m2*L*y(6)^2*sin(y(5))-(m1+m2)*g
 y(6)
 -g*L*cos(y(5))
 ];

% --------------------------------------------

function M = mass(t,y,m1,m2,L,g)
M = zeros(6,6);
M(1,1) = 1;
M(2,2) = m1 + m2;
M(2,6) = -m2*L*sin(y(5));
M(3,3) = 1;
```

```
M(4,4) = m1 + m2;
M(4,6) = m2*L*cos(y(5));
M(5,5) = 1;
M(6,2) = -L*sin(y(5));
M(6,4) = L*cos(y(5));
M(6,6) = L^2;
```

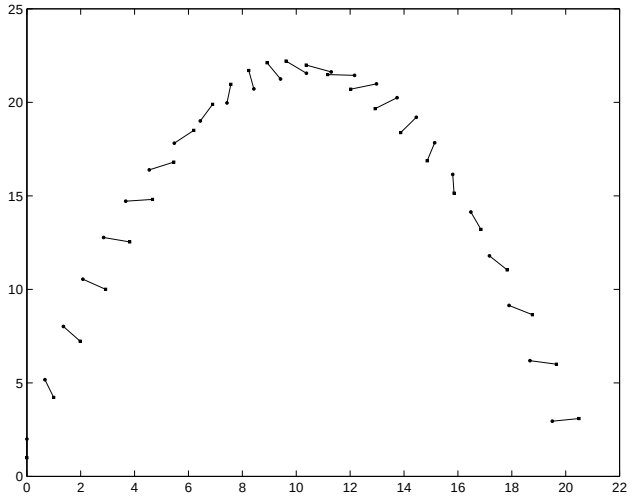

Figure 1.5: Solution of a thrown baton problem with mass matrix $M(t, y)$.

Example 1.3.7. The incompressible Navier–Stokes equations for time dependent fluid flow in one space dimension on an interval of length L can be formulated as the system of PDEs

$$\frac{\partial U}{\partial t} + A\frac{\partial U}{\partial z} = C$$

to be solved for $t \geq 0$ and $0 \leq z \leq L$. The vector $U = \begin{pmatrix} \rho \\ G \\ T \end{pmatrix}$ consists of three unknowns, the density ρ, the flow rate G, and the temperature T. In the system of PDEs, the vector

$$C = \begin{pmatrix} 0 \\ -KG\left|\dfrac{G}{\rho}\right| - \rho g_a \sin(\theta) \\ \dfrac{a^2 \Phi P_H \kappa}{C_p A_f} \end{pmatrix}$$

and the matrix

$$A = \begin{pmatrix} 0 & 1 & 0 \\ \dfrac{1}{\rho\kappa} - \dfrac{G^2}{\rho^2} & 2\dfrac{G}{\rho} & \dfrac{\beta}{\kappa} \\ -\dfrac{a^2\beta\overline{T}G}{\rho^2 C_p} & \dfrac{a^2\beta\overline{T}}{\rho C_p} & \dfrac{G}{\rho} \end{pmatrix}$$

Relevant fluid properties and other problem parameters are described in [127]. We use constant values that are found in `ch2ex4.m` with names that are obvious except possibly `sinth` for $\sin(\theta)$.

We also use $\overline{T} = T + 273.15$ and boundary conditions

$$\begin{aligned} \rho(0,t) &= \rho_0 = 795.5 \\ T(0,t) &= T_0 = 255.0 \\ G(L,t) &= G_0 = 270.9 \end{aligned}$$

We are interested in computing a steady state solution of these equations. At steady state we have $\rho_t = 0$, which implies that $G(z)$ is the constant G_0 on the whole interval $0 < z < L$ and the PDEs reduce to the ODEs

$$\begin{pmatrix} \frac{1}{\rho\kappa} - \frac{G_0^2}{\rho^2} & \frac{\beta}{\kappa} \\ -\frac{a^2\beta\overline{T}G_0}{\rho^2 C_p} & \frac{G_0}{\rho} \end{pmatrix} \begin{pmatrix} \frac{d\rho}{dz} \\ \frac{dT}{dz} \end{pmatrix} = \begin{pmatrix} -KG_0 \left| \frac{G_0}{\rho} \right| - \rho g_a \sin(\theta) \\ \frac{a^2 \Phi P_H \kappa}{C_p A_f} \end{pmatrix}$$

Converting the system of PDEs to a system of ODEs in this way is sometimes referred to as a Continuous Space, Discrete Time (CSDT) solution. Such a solution may result in an IVP for a system of ODEs or a BVP, depending on the boundary conditions for the PDEs. These equations are often used to model the subcooled liquid portion of a three phase steam generator in power systems. (Similar but more complicated equations apply to the saturated liquid–steam and pure steam regions.) In such a model the (moving) boundaries between the regions are determined using properties of the equation of state. For example, the ODEs may be integrated from $z = 0$ in the positive z direction until the density ρ is equal to the "liquid–side" saturation density $\rho_{\text{sat}}(T)$. As discussed in §1.3.1, we can find where this happens using an event function

$$g(z, \rho, T) = \rho(z) - \rho_{\text{sat}}(T(z))$$

Strictly for the purposes of illustration and convenience, we use a mockup of the equation of state:

$$\rho_{\text{sat}}(T) = -3.3(T - 290) + 738$$

Even with the gross simplifications of this model, formulating the IVP is somewhat complicated. However, once we have the IVP, it is easy enough to solve with one of the MATLAB IVP solvers because they accept problems formulated in terms of a mass matrix and they locate events. Invoking `ch2ex4.m` results in output to the screen of

```
Upper boundary at z = 2.09614.
```

and a figure shown here as Fig. 1.6. Exercise 1.32 modifies this program in a standard application of the steady state CSDT model to pump coast down.

```
function ch2ex4
% Define the physical constants:
global kappa beta a Cp K ga sinth Phi Ph Af G0
kappa = 0.171446272015689e-8;
beta  = 0.213024626664637e-2;
a     = 0.108595374561510e+4;
Cp    = 0.496941623289027e+4;
K     = 10;
ga    = 9.80665;
sinth = 1;
Phi   = 1.1e+5;
Ph    = 797.318;
Af    = 3.82760;
G0    = 270.9;
```

```
options = odeset('Mass',@mass,'MassSingular','no','Events',@events);
[z,y,ze,ye,ie] = ode45(@odes,[0 5],[795.5; 255.0],options);
if ~isempty(ie)
     fprintf('Upper boundary at z = %g.\n',ze(end));
end
plot(z,y);

%=====================================================================
function dydz = odes(z,y)
global kappa beta a Cp K ga sinth Phi Ph Af G0
rho = y(1);
T = y(2);
dydz = [ (-K*G0*abs(G0/rho) - rho*ga*sinth)
         (a^2 *Phi*Ph*kappa)/(Cp*Af)       ];

function A = mass(z,y)
global kappa beta a Cp K ga sinth Phi Ph Af G0
rho = y(1);
T = y(2);
A = zeros(2);
A(1,1) = 1/(rho*kappa) - (G0/rho)^2;
A(1,2) = beta/kappa;
A(2,1) = -(a^2 *beta*(T + 273.15)*G0)/(Cp*rho^2);
A(2,2) = G0/rho;

function [value,isterminal,direction] = events(z,y)
isterminal = 1;
direction  = 0;
rho   = y(1);
T     = y(2);
rhosat  = -3.3*(T - 290.0) + 738.0;
value = rho - rhosat;
```

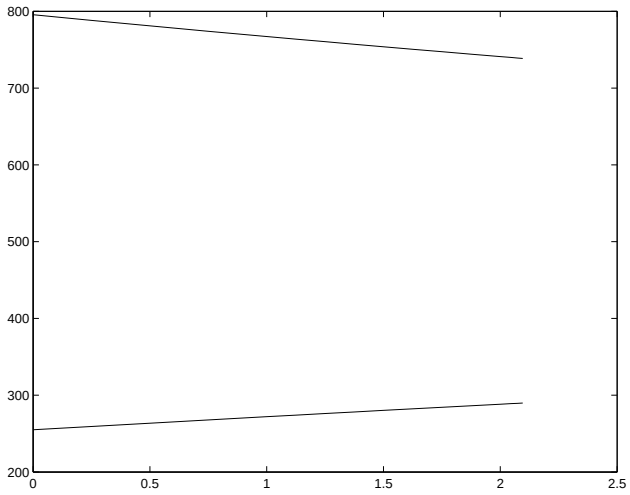

Figure 1.6: Steady solution to upper boundary of liquid region.

Exercise 1.30. The ODEs of the ozone model presented in Exercise 1.19 appear in a form appropriate for singular perturbation methods. This form involves a mass matrix $M = diag\{1, \epsilon\}$. Do

the exercise now with the ODEs formulated in terms of a mass matrix. In the computations, take advantage of M being constant.

Exercise 1.31. In the MATLAB demonstration program `batonode`, the length of the baton is 1 and the masses are both 0.1. Make yourself a copy of `batonode` and modify it to compute the motion when the length `L` is increased to 2 and one mass, say `m2`, is increased to 0.5. In this you will want to delete, or at least change, the setting of `axis`. What qualitative differences do you observe in the motion?

Exercise 1.32. A standard application of the steady state CSDT model of Example 1.3.7 is to pump coast down—the pumps are shut down and the inlet flow rate decreases exponentially fast. Solve the CSDT problem for `tD = linspace(0,1,11)` when the inlet flow rate at time `tD(i)` is

$$G_0 = 270.9\left(0.8 + 0.2e^{-\texttt{tD(i)}}\right)$$

You can do this by modifying the program `ch2ex4.m` so that it solves for the upper boundary `zU(i)` of the liquid region with this value of G_0 in a loop. The flow rate is already a global variable in `ch2ex4.m`, so you can define it in the loop just before calling `ode45`. Use `plot(tD,zU)` to plot the location of the upper boundary as a function of time.

Exercise 1.33. A double pendulum consists of a pendulum attached to a pendulum. Let $\theta_1(t)$ be the angle that the top pendulum makes with the vertical and $\theta_2(t)$, the angle the lower makes with the vertical. Correspondingly, let m_i be the masses of the bobs and L_i, the lengths of the pendulum arms. When the only force acting on the double pendulum is gravity with constant acceleration g, Borrelli and Coleman [21] develop the equations of motion

$$\begin{aligned}(m_1+m_2)L_1\theta_1'' + m_2L_2\cos(\theta_2-\theta_1)\theta_2'' \\ - m_2L_2(\theta_2')^2\sin(\theta_2-\theta_1) + (m_1+m_2)g\sin(\theta_1) = 0 \\ m_2L_2\theta_2'' + m_2L_1\cos(\theta_2-\theta_1)\theta_1'' \\ + m_2L_1(\theta_1')^2\sin(\theta_2-\theta_1) + m_2g\sin(\theta_2) = 0\end{aligned}$$

In an interesting exercise, Giordano and Weir [47] model a Chinook helicopter delivering supplies on two pallets slung beneath the helicopter as a double pendulum. They take the mass of the upper pallet to be $m_1 = 937.5$ slugs and the mass of the lower to be $m_2 = 312.5$ slugs. The lengths of the cables are $L_1 = L_2 = 16$ feet and $g = 32$ feet/second^2. The hook holding the lower pallet cable is bent open, and if the lower pallet oscillates through an arc of $\frac{\pi}{3}$ radians or more to the open end of the hook, the cable will come off the hook and the pallet will be lost. As a result of a sudden maneuver, the pallets begin moving subject to the initial conditions

$$\theta_1(0) = -0.5, \quad \theta_1'(0) = -1, \quad \theta_2(0) = 1, \quad \theta_2'(0) = 2$$

The differential equations are naturally formulated in terms of a mass matrix, so solve them in this form with `ode45` and default tolerances. Integrate from $t = 0$ to $t = 2\pi$, but terminate the run if the event function $\theta_2(t) - \frac{\pi}{3}$ vanishes, i.e., the lower pallet is lost, and report when this happens.

Giordano and Weir linearize the differential equations as

$$\begin{aligned}(m_1+m_2)L_1\theta_1'' + m_2L_2\theta_2'' + (m_1+m_2)g\theta_1 = 0 \\ m_2L_2\theta_2'' + m_2L_1\theta_1'' + m_2g\theta_2 = 0\end{aligned}$$

because it is not hard to solve this linear model analytically. The given values of the parameters satisfy $g = 2L_1 = 2L_2$ and $m_1 = 3m_2$. Because of this, the solution has a simple form

$$\begin{aligned}\theta_1(t) &= -\frac{1}{2}\cos(2t) - \frac{1}{2}\sin(2t) \\ \theta_2(t) &= \cos(2t) + \sin(2t)\end{aligned}$$

An analytical solution provides insight, but it is just as easy to solve the nonlinear model numerically as the linear model. Solve the linear model numerically and compare your numerical solution to this analytical solution. Compare your numerical solution and in particular, the time at which the lower pallet is lost, to the results you obtain with the nonlinear model. You should find that the results of the linear model are quite close to those of the nonlinear model.

1.3.3 Large Systems and the Method of Lines

Now we take up issues that are important when solving an IVP for a large system of equations. The MATLAB PSE is not appropriate for the very large systems solved routinely in some areas of general scientific computing, but it is possible to solve conveniently systems that are quite large. The *Method of Lines* (MOL) is a way of approximating PDEs by ODEs. It leads naturally to large systems of equations that may be stiff. The MOL is a popular tool for solving PDEs, but we discuss it here mainly as a source of large systems of ODEs. MATLAB itself has an MOL code called `pdepe` that solves small systems of parabolic and elliptic PDEs in one space variable and time. Also, the MATLAB PDE Toolbox uses the MOL to solve PDEs in two space variables and time.

The idea of the MOL is to discretize all but one of the variables of a PDE so as to obtain a system of ODEs. This approach is called a *semi–discretization*. Typically the spatial variables are discretized and time is used as the continuous variable. There are several popular ways of discretizing the spatial variable. Two will be illustrated with an example taken from Trefethen [128].

Example 1.3.8. The one–way wave (also known as the advection or convection) PDE

$$u_t + c(x)u_x = 0, \qquad c(x) = \frac{1}{5} + \sin^2(x-1)$$

is solved for $0 \le x \le 2\pi$ and $0 \le t \le 8$. Trefethen [128] considers initial values $u(x,0) = e^{-100(x-1)^2}$ and periodic boundary conditions, $u(0,t) = u(2\pi,t)$. As he notes, the initial profile $u(x,0)$ is not periodic, but it decays so fast near the ends of the interval that it can be regarded as periodic. Also, solutions of this PDE are waves that move to the right. As seen in Fig. 1.7, the peak that is initially at $x = 1$ does not reach the boundary of the region in this time interval. For later use we note that a boundary condition must be specified at the left end of the interval, but none is needed at the right. In the MOL, a grid $x_1 < x_2 < \cdots < x_N$ is chosen in the interval $[0, 2\pi]$ and functions $v_m(t)$ are used to approximate $u(x_m, t)$ for $m = 1, 2, \ldots, N$. These functions are determined by a system of ODEs

$$\frac{dv_m}{dt} = -c(x_m)(Dv)_m$$

with initial values

$$v_m(0) = u(x_m, 0)$$

for $m = 1, 2, \ldots, N$. If $(Dv)_m \approx u_x(x_m, t)$, this is an obvious approximation of the PDE. Trefethen's delightful little book is about spectral methods. In that approach, the partial derivative is approximated at (x_m, t) by interpolating the function values $v_1(t), v_2(t), \ldots, v_N(t)$ with a trigonometric polynomial in x, differentiating this polynomial with respect to x, and evaluating this derivative at x_m. It is characteristic of a spectral method that derivatives are approximated using data from the whole interval. When the mesh points are equally spaced, this kind of approximation is made practical using the fast Fourier transform (FFT). Just how this is done is not important here and the details can be found in the discussion of Trefethen's program `p6`. Our program `ch2ex5.m` is much like `p6` except that Trefethen uses a constant step method with memory to integrate the ODEs and we simply use `ode23`. The general purpose IVP solver is a little more expensive to use, but it avoids the issue of starting a method with memory, adapts the time steps to the solution, and controls the error of the time integration. And, someone else has gone to all the trouble of writing a quality program to do the time integration!

```
function ch2ex5
global ixindices mc
```

```
% Spatial grid, initial values:
N = 128;
x = (2*pi/N)*(1:N);
v0 = exp(-100*(x - 1) .^ 2);

% Quantities passed as global variables to f(t,v):
ixindices = i * [0:N/2-1 0 -N/2+1:-1]';
mc = - (0.2 + sin(x - 1) .^ 2)';

% Integrate and plot:
t = 0:0.30:8;
[t,v] = ode23(@f,t,v0);
mesh(x,t,v), view(10,70), axis([0 2*pi 0 8 0 5])
ylabel t, zlabel u, grid off

%=============================================================
function dvdt = f(t,v)
global ixindices mc
dvdt = mc .* real(ifft(ixindices .* fft(v)));
```

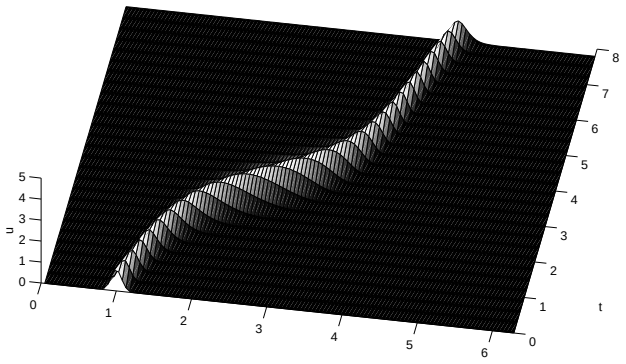

Figure 1.7: Solution of $u_t + c(x)u_x = 0$ with the MOL.

When using a constant step size as Trefethen does, a key issue is selecting a time step small enough to compute an accurate solution. When using a solver like `ode23`, the issue is choosing output points that result in an acceptable plot because the solver selects time steps that provide an accurate solution and returns answers wherever we like. It would be natural to use the default output at every step, but for a large system, storage can be an issue. You can reduce the storage required by asking for answers at specific points. Alternatively, you can gain complete control over the amount of output by writing an output function as illustrated in Example 1.3.3. In `ch2ex5.m` we specify half as many output points as Trefethen because this is sufficient to provide an acceptable plot and it significantly reduces the size of the output file. For just this reason the PDE solver `pdepe` requires you to specify not only the fixed mesh in the spatial variable, but also the times at which you want output. In a situation like this, you should not ask for output in the form of a solution structure because the structure contains not only the solution at every step, but considerably more information that is needed for evaluating the continuous extension.

It is commonly thought that any system of ODEs with a moderate to large number of unknowns that arises from MOL is stiff, but the matter is not that simple. The system of 128 ODEs arising

in the spectral discretization of this wave equation is at most mildly stiff. This is verified by using the stiff solver `ode15s` instead of the non–stiff solver `ode23`. The change *increases* the run time by more than a factor of 5. When no substantial advantage is to be gained from the superior stability of its formulas, `ode15s` is often (much) more expensive than one of the solvers intended for non–stiff IVPs. By default `ode15s` approximates the Jacobian matrix numerically. This is expensive because for a system of N equations, it makes N evaluations of the ODE function $f(t, v)$ for this purpose (and a few more if it has doubts about the quality of the approximation). `ode15s` handles this much more efficiently than most stiff solvers because it saves Jacobians and forms new ones only when it believes this is necessary for the efficient evaluation of the implicit formulas. When solving non–stiff problems, the step size is generally small enough that only a few Jacobians are formed. To explore this, we set the option `Stats` to `on` and ran the modified `ch2ex5.m` to obtain

```
>> ch2ex5
199 successful steps
6 failed attempts
328 function evaluations
1 partial derivatives
34 LU decompositions
324 solutions of linear systems
```

(*LU decompositions* is a synonym for *LU factorizations*.) The implicit formulas are evaluated with a remarkably small number of function evaluations per step. The solver made only one Jacobian (partial derivative) evaluation and reused it each of the 34 times it formed and factored an iteration matrix. By these measures `ode15s` solves the non–stiff IVP quite efficiently. Indeed, `ode23` took 315 steps and made 946 function evaluations. However, `ode15s` is slower because it had to solve many systems involving 128 linear equations. They are solved very efficiently in MATLAB, but the extra overhead of the BDF code makes it less efficient than this Runge–Kutta code for this problem. On the other hand, the higher order explicit Runge–Kutta code `ode45` is less efficient than `ode15s`. `ode45` is less efficient for this problem than `ode23` because its formulas have smaller stability regions.

Example 1.3.9. The spectral approximation of u_x in Example 1.3.8 is very accurate when the function $u(x, t)$ is smooth. Finite difference approximations are a more common alternative. They are of (much) lower order of accuracy, but are local in nature, so require (much) less smoothness of the solution. It is worthy of comment that the one–way wave equation can have solutions with discontinuities in the spatial variable, a matter that receives a great deal of attention in the numerical solution of such PDEs. A simple, yet reasonable, finite difference scheme for this equation is a first order upwind difference approximation to the spatial derivative. Again we use a mesh in x of N points with equal spacing of h. Because $c(x) > 0$, the wave is moving to the right and the scheme results in

$$\frac{dv_m}{dt} = -c(x_m)\left(\frac{v_m - v_{m-1}}{h}\right)$$

Trefethen assumed periodic boundary conditions because it simplified the use of spectral approximations. At the left boundary this assumption leads to

$$\frac{dv_1}{dt} = -c(x_1)\left(\frac{v_1 - v_N}{h}\right) \tag{1.38}$$

We'll return to this matter, but right now let us also change the boundary condition to simplify the application of the finite difference scheme. With the given initial data, it is just as plausible to use the boundary condition $u_x(0, t) = 0$. This boundary condition is approximated by $v_1'(t) = 0$. With a first order approximation to $u_x(x, t)$, we need a fine mesh to obtain a numerical solution comparable to that of Fig. 1.7. The program `ch2ex6.m` approximates this figure by taking $N = 1000$. Notice that with such a fine mesh, we do not plot the solution at all the mesh points.

The finite difference program is straightforward, but new issues arise when we solve problems involving a large number of equations (here 1000). The problems resulting from MOL with a fine space discretization can be stiff. Indeed, the parabolic PDEs solved with `pdepe` are stiff when the

spatial mesh is at all fine. The IVP of this example is not stiff, but it is simple enough that it shows clearly how to proceed when we treat it as a stiff problem. If we solve it with a code intended for stiff problems, the code must form Jacobians. With N equations, this matrix is $N \times N$. Thousands of equations are common in this context, making it important to account for the sparsity of the equations, both with respect to storage and the efficiency of solving linear systems. Sparse matrix technology is fully integrated into the MATLAB problem solving environment, but it is not so readily available in general scientific computing. In fact, the only provision for sparsity in a typical stiff solver for general scientific computing is for *banded Jacobians*. (A banded matrix is one that has all its non–zero entries confined to a band of diagonals about the principal diagonal.) Somehow we must inform the solver of the zeros in the Jacobian $J = (\frac{\partial f_m}{\partial v_k})$. The simplest way is to provide the Jacobian matrix analytically as a sparse matrix. This is done much as with mass matrices by means of an option called `Jacobian`. If the Jacobian is a constant matrix, you should provide it as the value of this option. This could have been done in `ch2ex6.m` with

```
B = [ [-c_h(2:N); 0] [0; c_h(2:N)] ];
J = spdiags(B,-1:0,N,N);
options = odeset('Jacobian',J);
```

When the Jacobian is not constant, you provide a function that evaluates the Jacobian and returns a sparse matrix as its value. In either case, the solvers recognize that the Jacobian is sparse and deal with it appropriately. When it is not too much trouble to work out an analytical expression for the Jacobian, it is best to do so because approximating Jacobians numerically is a relatively expensive and difficult task. Also, the methods are generally somewhat more efficient when using analytical Jacobians. Still, the default in the MATLAB solvers is to approximate Jacobians numerically because it is so convenient. For large problems it is then very important that you inform the solver of the structure of the Jacobian. The entry $J_{m,k}$ is identically zero when equation m does not depend on component k of the solution. You inform the solver of this by creating a sparse matrix S that has zeros in the places where the Jacobian is identically zero and ones elsewhere. It is provided as the value of the option `JPattern`. This is done in a very straightforward way in `ch2ex6.m`. The example shows how you might proceed for more complicated patterns, but for this particular set of ODEs the non–zero entries occur on diagonals, making it natural to define the pattern with `spdiags`. This could have been done in `ch2ex6.m` with

```
S = spdiags(ones(N,2),-1:0,N,N))};
S(1,1) = 0
```

You are asked in Exercise 1.34 to verify that this IVP is not stiff. Nonetheless, we solve it with `ode15s` in `ch2ex6.m` to show how to solve a large stiff problem. The Jacobian is approximated by finite differences, but the sparsity pattern is supplied to make this efficient. It would be more efficient to provide an analytical expression for the constant Jacobian matrix as the value of the `Jacobian` option.

```
function ch2ex6
global c_h

% Spatial grid, initial values:
N = 1000;
h = 2*pi/N;
x = h*(1:N);
v0 = exp(-100*(x - 1) .^ 2);
c_h = - (0.2 + sin(x - 1) .^ 2)' / h;

% Sparsity pattern for the Jacobian:
S = sparse(N,N);
for m = 2:N
    S(m,m-1) = 1;
```

```
    S(m,m) = 1;
end
options = odeset('JPattern',S);

% Integrate and plot:
t = 0:0.30:8;
[t,v] = ode15s(@f,t,v0,options);
pltspace = ceil(N/128);
x = x(1:pltspace:end);
v = v(:,1:pltspace:end);
surf(x,t,v), view(10,70), axis([0 2*pi 0 8 0 5])
ylabel t, zlabel u, grid off

%===============================================
function dvdt = f(t,v)
global c_h
dvdt = c_h .* [0; diff(v)];
```

Now we can understand better the role of the boundary conditions. With periodic boundary conditions, this example does not have a banded Jacobian because the first equation depends on v_N. Because of this, typical codes for stiff problems in general scientific computing cannot solve this IVP directly. In contrast, with the stiff solvers of MATLAB, all we must do is follow the definition of `S` in `ch2ex6.m` with the two statements

```
S(1,1) = 1;
S(1,N) = 1;
```

Exercise 1.35 asks you to solve this IVP with periodic boundary conditions.

If no attention is paid to sparsity, forming an approximate Jacobian by finite differences costs (at least) one evaluation of the ODE function $f(t, v)$ for each of the N columns of the matrix. The MATLAB solvers use a development of an algorithm due to Curtis, Powell, and Reid [30] that can reduce this cost greatly by taking into account entries of the Jacobian that are known to be zero. For instance, if the Jacobian is banded and has D diagonals, this algorithm evaluates the Jacobian with only D evaluations of the ODE function $f(t, v)$, *no matter what the value of N*. Informing the solver of the structure of the Jacobian in `ch2ex6.m` allows it to approximate the Jacobian in only *two* evaluations of the ODE function $f(t, v)$ instead of the $D = N = 1000$ evaluations required if the structure is ignored. Clearly this is a matter of the greatest importance for efficiency, quite aside from its importance to reducing the storage and the cost of solving the linear systems involving the iteration matrix.

Example 1.3.10. As we have just seen, one way to speed up the numerical approximation of Jacobians is to use sparsity to reduce the number of function evaluations. Another is to evaluate the functions faster. Vectorization is a very valuable tool for speeding up MATLAB programs and often it is not much work to vectorize a program. When the solvers approximate a Jacobian, they evaluate $f(t, y)$ for several vectors y at the same value of t. This can be exploited by coding the function so that if it is called with arguments `(t,[y1 y2 ... ])`, it will compute and return `[f(t,y1) f(t,y2) ... ]`. You tell the solver that you have done this by setting the option `Vectorized` to `on`. Because all the built–in functions of MATLAB are vectorized, vectorizing $f(t, y)$ is often a matter of treating `y` as an array of column vectors and putting a dot before operators to make them apply to arrays. The brusselator problem is a pair of coupled PDEs solved in Hairer and Wanner [52, pp. 6–8] by the MOL with finite difference discretization of the spatial variable. It is also solved by one of the demonstration programs included in MATLAB. When invoked as `brussode(N)`, this program forms and integrates a system of $2N$ ODEs. The default value of the parameter N is 20. Here we consider just one line of the program to make the point about vectorizing $f(t, y)$. Some of the components of `dydt` are evaluated in a loop on `i` that might have been coded as

```
dydt(i) = 1 + y(i+1)*y(i)^2 - 4*y(i) + ...
              c*(y(i-2) - 2*y(i) + y(i+2));
```

This processes a column vector `y(:)`. A vector version must perform the same operations on the columns of an array `y(:,:)`. It is coded in `brussode` as

```
dydt(i,:) = 1 + y(i+1,:).*y(i,:).^2 - 4*y(i,:) + ...
                c*(y(i-2,:) - 2*y(i,:) + y(i+2,:));
```

The dot before the multiplication and exponentiation operators makes the operations apply by components to arrays. Adding the scalar 1 to an array is interpreted as adding it to each entry of the array. This is easy enough, but in point of fact, vectorization is not important for this example. The sparsity structure of the Jacobian is such that only a few function evaluations are needed for each Jacobian and only a few Jacobians are formed. Indeed, by setting the option `Stats` to `on` we found that only two Jacobians were formed for the whole integration. Whether vectorization is helpful depends on the sparsity pattern of the Jacobian and how expensive it is to evaluate the function. A similar vectorization is usually quite helpful when solving BVPs with `bvp4c` because of its numerical method, so we return to this matter in Chapter **??**.

To illustrate a vectorization that is not so straightforward, suppose that we replace `ode23` in `ch2ex5.m` by `ode15s` and so wish to vectorize

```
dvdt = mc .* real(ifft(ixindices .* fft(v)));
```

The fast Fourier transform routines are vectorized, so `fft(v)` works fine when `v` has more than one column. We get into trouble when we form `ixindices .* fft(v)` because `ixindices` is a column vector and the array sizes do not match up. We need to do this array multiplication on each of the columns of `fft(v)`. We could do this in a loop. In such a loop we must take account of the fact that the solver calls the function with an array `v` that generally has `N` columns, but sometimes only one. A more efficient way to evaluate the derivatives is to make `ixindices` a matrix with `N` columns, all the same. This is done easily with `repmat`:

```
temp1 = i * [0:N/2-1 0 -N/2+1:-1]';
ixindices = repmat(temp1,1,N);
```

To match up the dimensions properly for the array multiplication of `ixindices` and `fft(v)`, we just use `size` to find out how many columns there are in `v` and use the corresponding number of columns of `ixindices` in the multiplication. The same change must be made with the array `mc`. The evaluation of `dvdt` then becomes

```
nc = size(v,2);
dvdt = mc(:,1:nc) .* real(ifft(ixindices(:,1:nc) .* fft(v)));
```

Vectorizing `f(t,v)` has little effect for this particular IVP because it is not stiff and `ode15s` forms only one Jacobian. However, the point of this example is to show how complications can arise in vectorization.

Example 1.3.11. Mass matrices arise naturally when a Galerkin method is used for the spatial discretization. To illustrate this and so introduce a discussion of associated issues for large stiff systems, we consider an example of Fletcher [43]. He solves the heat equation,

$$u_t = u_{xx}, \quad 0 \le x \le 1$$

with initial value

$$u(x,0) = x + \sin(\pi x)$$

and boundary values

$$u(0,t) = 0, \quad u(1,t) = 1$$

An approximate solution is sought in the form

$$v(x,t) = \sum_m S_m(x) v_m(t)$$

For a given value of t, this approximation satisfies the PDE with a residual

$$R(x,t) = v_t(x,t) - v_{xx}(x,t)$$

An inner product of two functions continuous on $[0,1]$ is defined by

$$(f,g) = \int_0^1 f(x)g(x)\,dx$$

The Galerkin method projects $u(x,t)$ into the space of functions of the form $v(x,t)$ by requiring the residual of the approximation to be orthogonal to the basis functions $S_k(x)$, i.e.,

$$(R, S_k) = (v_t, S_k) - (v_{xx}, S_k) = 0$$

for each k. If each shape function $S_m(x)$ is a piecewise linear function with $S_m(x_m) = 1$ and $S_m(x_j) = 0$ for $j \neq m$, then

$$v(x_m,t) = v_m(t) \approx u(x_m,t)$$

Using the form of $v(x,t)$ and performing an integration by parts leads to the equation

$$\sum_m (S_m, S_k) \frac{dv_m}{dt} + \sum_m \left(\frac{dS_m}{dx}, \frac{dS_k}{dx} \right) v_m = 0$$

With a constant mesh spacing h, working out the inner products leads to the ODEs

$$\frac{1}{6}\frac{dv_{m-1}}{dt} + \frac{4}{6}\frac{dv_m}{dt} + \frac{1}{6}\frac{dv_{m+1}}{dt} = \frac{v_{m-1} - 2v_m + v_{m+1}}{h^2}$$

for $m = 1, 2, \ldots, N$. From the boundary conditions we have

$$v_0(t) = 0,\ v_{N+1}(t) = 1$$

These quantities appear only in the first and last ODEs. Notice that the last ODE is inhomogeneous for this reason. From the initial condition, $v_m(0) = u(x_m, 0)$ for $m = 1, 2, \ldots, N$.

Many problems are naturally formulated in terms of mass matrices. If $M(t,y)$ does not depend strongly on y, it is not difficult to modify the numerical methods to take it into account and the presence of a mass matrix has little effect on the computation. For a large system it is essential to account for the structure of the mass matrix as well as the structure of the Jacobian. Although it is often both convenient and efficient to use the standard form (1.37), the typical general–purpose BDF code in general scientific computing does not provide for mass matrices. Storage and the user interface are important reasons for this. For example, when the mass matrix is constant, the iteration matrix for the BDFs is changed from $I - h\gamma J$ to $M - h\gamma J$. The change complicates specification of the matrices and management of storage because the structures of M and J must be merged in forming the iteration matrix. This complication is not present in the MATLAB IVP solvers because the matrices are provided as general sparse matrices and the PSE deals automatically with the storage issues.

The modifications to the algorithms due to the presence of a mass matrix depend on its form. The weaker the dependence of $M(t,y)$ on y, the better. The most favorable case is a constant mass matrix. The solver is told of this by providing the matrix as the value of the `Mass` option. If the mass matrix is not constant, the option `MStateDependence` is used to inform the solver about how strongly $M(t,y)$ depends on y. This is an issue only for moderate to large systems. There are three values possible for this option. The most favorable is `none`, indicating a mass matrix that is a function of t only, i.e., the matrix has the form $M(t)$. The default value is `weak`. The most difficult

in both theory and practice is `strong`. If $M(t, y)$ depends strongly on y, the standard form (1.37) is much less advantageous and the computation resembles much more closely what must be done for a general implicit system of the form

$$F(t, y, y') = 0 \tag{1.39}$$

Solving efficiently a large system of ODEs when the mass matrix has a strong state dependence is somewhat complicated and the default of a `weak` state dependence is usually satisfactory, so we leave this case to the MATLAB documentation.

There are a number of codes in general scientific computing that accept problems of the form (1.39), `DASSL` [18] being a popular one, and a future version of MATLAB will include a code for such problems. This general form includes differential algebraic equations (DAEs). They can be much more difficult than ODEs in both theory and practice. An obvious theoretical difficulty is that an initial value $y(t_0)$ may not be sufficient to specify a solution—we must find an initial slope $y'(t_0)$) that is *consistent* in the sense that equation (1.39) is satisfied with arguments t_0, $y(t_0)$, and $y'(t_0)$. It is pretty obvious from the form of the equation that the solvers need $\frac{\partial F}{\partial y'}$ as well as $\frac{\partial F}{\partial y}$, complicating the user interface and storage management.

For the present example it is convenient to supply both the mass matrix and the Jacobian matrices directly because they are constant. Indeed, it is convenient to use the Jacobian matrix in evaluating the ODEs. Fig. 1.8 shows the output of `ch2ex7.m`, but a better understanding of the solution is obtained by using the `Rotate 3D` tool to examine the surface from different perspectives.

```
function ch2ex7
global J h

N = 20;
h = 1/(N+1);
x = h*(1:N);
v0 = x + sin(pi*x);
e = ones(N,1);
M = spdiags([e  4*e e],-1:1,N,N)/6;
J = spdiags([e -2*e e],-1:1,N,N)/h^2;
options = odeset('Mass',M,'Jacobian',J);
[t,v] = ode15s(@f,[0 0.5],v0,options);
% Add the boundary values:
x = [0 x 1];
npts = length(t);
v = [zeros(npts,1) v ones(npts,1)];
surf(x,t,v)

%=======================================
function dvdt = f(t,v)
global J h
dvdt = J*v;
dvdt(end) = dvdt(end) + 1/h^2;
```

Exercise 1.34. Modify `ch2ex6.m` to use `ode23`. Using `tic` and `toc`, measure how much the run time is affected by this change, hence how stiff the IVP appears to be. It can't be very stiff if you can solve it at all with `ode23`.

Exercise 1.35. Example 1.3.9 points out that it is easy to deal with periodic boundary conditions because `ode15s` provides for general sparse matrices. See for yourself by modifying `ch2ex6.m` so as to solve the problem with periodic boundary conditions. The example explains how to form the sparsity pattern S in this case. Only the first component in `dvdt` must be altered so that it corresponds to the periodic boundary condition (1.38).

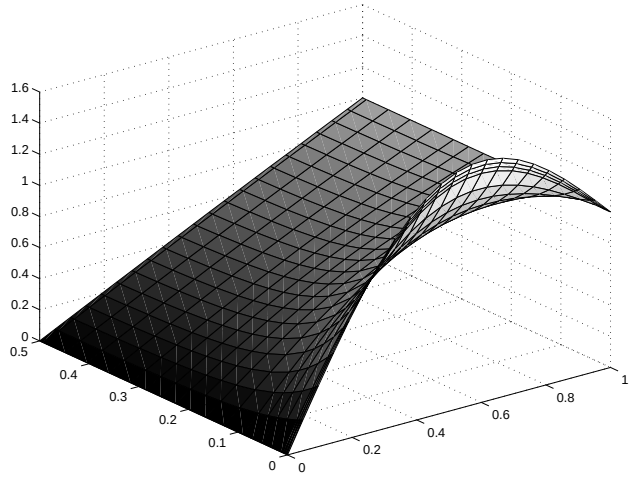

Figure 1.8: Solution of the differential equation $u_t = u_{xx}$.

Exercise 1.36. The quench front problem of [74] models a cooled liquid rising on a hot metal rod by the PDE

$$u_t = u_{xx} + g(u)$$

for $0 \le x \le 1,\ 0 < t$. Here

$$g(u) = \begin{cases} -Au & \text{if} \quad u \le u_c \\ 0 & \text{if} \quad u_c < u \end{cases}$$

with $A = 2 \cdot 10^5$ and $u_c = 0.5$. The boundary conditions are $u(0,t) = 0$ and $u_x(1,t) = 0$. The initial condition is

$$u(x,0) = \begin{cases} 0 & \text{if} \quad 0 \le x \le 0.1, \\ \left(\dfrac{x - 0.1}{0.15}\right) & \text{if} \quad 0.1 < x < 0.25, \\ 1 & \text{if} \quad 0.25 \le x \le 1 \end{cases}$$

We'll approximate the solution of this PDE by the MOL with finite differences. The computations will help us understand stiffness.

For an integer N, let $h = \frac{1}{N}$ and define the grid $x_m = mh$ for $m = 0, 1, \ldots, N+1$. We approximate the PDE by letting $v_m(t) \approx u(x_m, t)$ and discretizing in space with central differences to obtain

$$\frac{dv_m}{dt} = \frac{v_{m+1} - 2v_m + v_{m-1}}{h^2} + g(v_m)$$

As written, the equation for $m = 1$ requires $v_0(t)$. To satisfy the boundary condition $u(0,t) = 0$, we define $v_0(t) = 0$ and eliminate it from this equation. Similarly, the equation for $m = N$ requires $v_{N+1}(t)$. We approximate the boundary condition at $x = 1$ by central differences as

$$0 = u_x(1,t) \approx \frac{v_{N+1} - v_{N-1}}{2h}$$

Accordingly, we take $v_{N+1} = v_{N-1}$ and eliminate it in the equation for $m = N$. Initial values for the ODEs are provided by $v_m(0) = u(x_m, 0)$. Solve this IVP for the equations $m = 1, 2, \ldots, N$ with $N = 50$. In solving this problem, take advantage of the fact that the Jacobian J is a constant, sparse matrix. You can do this as in `ch2ex7.m`, though here $J_{N,N-1} = 2h^{-2}$. It is convenient and efficient to use J when evaluating the ODEs. You might also vectorize the evaluation of the $g(v_m)$. In this you might find it helpful to ask yourself what is the result of `any(w <= 0,2)` when `w` is a column vector. Control the output by taking `tspan = 0:0.001:0.006`. If you then plot the computed solution `v` against the mesh `x`, you will obtain solution profiles that show how the solution evolves in time. Solve the IVP using each of `ode15s` and `ode23`. Use `tic` and `toc` to measure

the costs of integration. You will find that this IVP is not stiff—`ode23` is much more efficient than `ode15s`. People familiar with the MOL are likely to find this surprising. To see what they might have expected, drop the inhomogeneous term $g(u)$ from the ODEs and solve with `tspan = 0:0.1:0.6`. You will find that this new IVP is (moderately) stiff.

What's going on? The inhomogeneous term $g(u)$ is not smooth. This results in solution components $v_m(t)$ that are not smooth as they pass through the value u_c. For this problem that happens only once for any m. The solver has to locate this point and use a small step size to pass it, but an isolated point where a solution is not smooth has little impact on an integration. The snag here is that this happens at different times for the different components, and there are a lot of components. Remember, for an IVP to be stiff, the solution must be smooth so that the step size of an explicit method is limited by stability. Stability does not limit the step size significantly for the quench problem, so `ode15s` is much less efficient than `ode23` even though we have a constant, analytical Jacobian.

1.3.4 Singularities

Standard codes for IVPs expect some smoothness and when it is not present at an isolated point, we must supplement the codes with some analytical work. The idea is to approximate the solution of interest near the singular point with a series or asymptotic expansion and approximate it elsewhere with a standard IVP solver. The expression "solution of interest" alludes to the possibility of more than one solution at a singular point. Singularities are much more common for BVPs, in part because problems are often set on an infinite interval, so we show here how to proceed with a single example and a couple of exercises and return to the matter in Chapter **??** where many examples are discussed at length.

Example 1.3.12. A classical analysis [76, p. 103 ff.] of the collapse of a spherical cavity in a liquid leads to the IVP

$$(y')^2 = \frac{2}{3}(y^{-3} - 1), \qquad y(0) = 1 \tag{1.40}$$

The non–dimensional variables here are the time x and the radius of the cavity y. The integration is to terminate when y vanishes, representing total collapse of the cavity. In standard form the ODE is

$$\frac{dy}{dx} = -\sqrt{\frac{2}{3}(y^{-3} - 1)}$$

where the minus sign is taken because the equation is to model *collapse* of the cavity. This IVP presents two kinds of difficulties. The existence and uniqueness result for IVPs supposes that the ODE function f in $y' = f(x, y)$ is smooth in a region containing the initial point $(0, 1)$. That is not the case here because the initial point is on the boundary of the region where f is defined. This leaves open the possibility of more than one solution and in fact, there is an obvious solution $y(x) \equiv 1$ in addition to the decreasing solution that we expect physically. Certainly we must consider how to compute the "right" solution. The other difficulty is that at the time of total collapse, x_c, we have $y(x_c) = 0$ and we find from the ODE that $y'(x_c) = -\infty$.

We approximate the solution $y(x)$ for $0 \le x \le d$ with a Taylor series. We then use $y_d \approx y(d)$ computed in this way to start a numerical integration for $x \ge d$ where the ODE is not singular. The symbolic capabilities of MATLAB make it easy to obtain the series. Taking into account the initial value, we look for an approximation of the form $y(x) = 1 + ax + bx^2 + \ldots$. The script

```
syms y x a b res
y = 1 + a*x + b*x^2;
res = taylor(diff(y)^2 - (2/3)*(1/y^3 - 1),3)
```

substitutes this form into the ODE and computes the first three terms of a Taylor expansion of the residual. The result of this computation is

```
res = a^2+(4*a*b+2*a)*x+(4*b^2+2*b-4*a^2)*x^2
```

To satisfy the ODE as well as possible, we must choose the coefficients so as to make as many terms as possible vanish, starting from the lowest order. To remove the constant term in the residual, we must take $a = 0$, which also removes the term in x. To remove the term in x^2, we have two choices for b, namely $b = 0$ and $b = -0.5$. Accordingly, this singular IVP appears to have exactly two solutions that can be expanded in Taylor series. One choice for b corresponds to the solution $y(x) \equiv 1$ that we have already recognized. The other solution

$$y(x) = 1 - \frac{1}{2}x^2 + \dots$$

is decreasing for small x. It appears then that there is a unique solution with the expected physical behavior. Continuing in this way it is found easily that

$$y(x) = 1 - \frac{1}{2}x^2 - \frac{1}{6}x^4 - \frac{19}{180}x^6 + \dots$$

In `ch2ex8.m` we approximate $y(x)$ by the first three terms of this series and estimate the error of the approximation by the next term. In this way we estimate that the relative error of this approximation to $y(d)$ is about 10^{-7} when $d = 0.1$.

The difficulty with calculating the vertical slope at the end of the integration is resolved by interchanging the independent and dependent variables. This is a technique familiar in the theory of ODEs that can be quite useful numerically. We must be careful that interchanging variables is permissible. We can use y as the independent variable if we stay away from the initial point because the ODE shows that $y(x)$ is strictly decreasing. We cannot use y as the independent variable there because $y'(x)$ vanishes at $x = 0$ and we'd have the same problem of a vertical slope after interchanging variables. For this reason we integrate

$$\frac{dx}{dy} = -\sqrt{\frac{3y^3}{2(1-y^3)}}$$

with initial value $x = d$ from $y = y_d$ to $y = 0$. The time of total collapse is the value of x at $y = 0$. After this analytical preparation of the problem, it is easily solved as in `ch2ex8.m`. To plot the solution over the whole interval, we must augment the numerical solution with values obtained by evaluating the series on the interval $[0, d]$. For this particular problem, a smooth graph is obtained by adding only one value, the initial value $y(0) = 1$. You might wish to see what happens if you make the initial point d much larger or much smaller than $d = 0.1$.

```
function ch2ex8
d = 0.1;
yd = 1 - (1/2)*d^2 - (1/6)*d^4;
erryd = (19/180)*d^6;
fprintf('At d = %g, the error in y(d) is about %5.1e.\n',d,erryd);

[y,x] = ode45(@ode,[yd 0],d);
fprintf('Total collapse occurs at x = %g.\n',x(end));

% Augment the arrays with y = 1 at x = 0 for the plot
% and plot with the original independent variable x.
y = [1; y];
x = [0; x];
plot(x,y)

%=======================================================
function dxdy = ode(y,x)
dxdy = - sqrt(3*y^3/(2*(1 - y^3)));
```

In addition to Fig. 1.9 this program results in the output

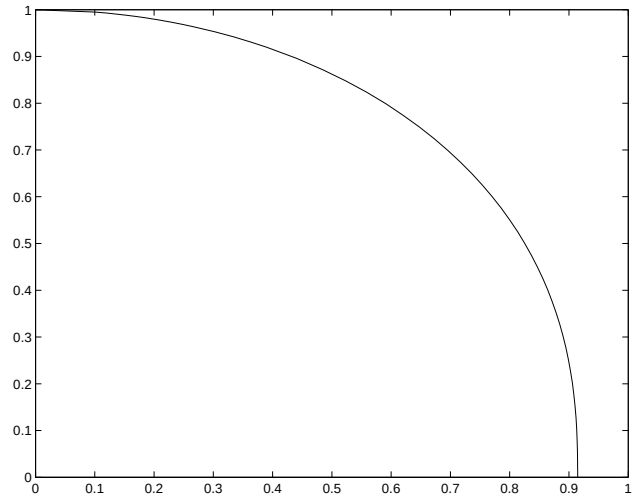

Figure 1.9: Solution of the cavity collapse problem.

```
>> ch2ex8
At d = 0.1, the error in y(d) is about 1.1e-007.
Total collapse occurs at x = 0.914704
```

Exercise 1.37. H.T. Davis [31, p. 371 ff.] discusses the use of Emden's equation

$$\frac{d^2y}{dx^2} + \frac{2}{x}\frac{dy}{dx} + y^n = 0$$

for modeling the thermal behavior of a spherical cloud of gas. Here n is a physical parameter. The singular coefficient arises when a PDE modeling the thermal behavior is reduced to an ODE by exploiting the spherical symmetry. Symmetry implies that the initial value $y'(0) = 0$ and in these non–dimensional variables, $y(0) = 1$. We expect a solution that is bounded and smooth at the origin, so we expect that we can expand the solution in a Taylor series. Derive the series solution reported by Davis,

$$y(x) = 1 - \frac{x^2}{3!} + n\frac{x^4}{5!} + (5n - 8n^2)\frac{x^6}{3 \cdot 7!} + \ldots$$

When the cloud of gas is a star, the first zero of the solution represents the radius of the star. Davis uses $n = 3$ in modeling the bright component of Capella and reports the first zero to be at about $x = 6.9$. You are to confirm this result. Use the first three terms of the series to compute an approximation to $y(0.1)$ and the derivative of these terms to approximate $y'(0.1)$. Estimate the error of each of these approximations by the magnitude of the first term neglected in the series. Using the analytical approximations to $y(0.1)$ and $y'(0.1)$ as the initial values, integrate the ODE with `ode45` until the terminal event $y(x) = 0$ occurs. You'll need to guess how far to go, so be sure to check that the integration terminates because of an event. The approximate solution at $x = 0.1$ is so accurate that it is reasonable to integrate with tolerances more stringent than the default values and so be more confident in the location of the first zero of $y(x)$. You might, e.g., use a relative error tolerance of 10^{-8} and an absolute error tolerance of 10^{-10}.

Exercise 1.38. Kamke [66, p. 598] states that the IVP

$$y(y'')^2 = e^{2x}, \qquad y(0) = 0,\ y'(0) = 0$$

describes space charge current in a cylindrical capacitor. He suggests approximating the solution with a series of the form

$$y(x) = x^p(a + bx + \ldots)$$

Work out p and a. Argue that the IVP has only one solution of this form. Because it is singular at the origin, we must supplement numerical integration of the ODE with this analytical approximation for small x. For this you are given that

$$b = \frac{27}{40}a^{-2}$$

If you have not done Exercise **??**, which asks you to formulate the ODE as a pair of explicit first order systems, you should do it now. In the present exercise you are to solve numerically the system for which y' is increasing near $x = 0$. Evaluate the series and its derivative to obtain approximations to $y(x_0)$ and $y'(x_0)$ for $x_0 = 0.001$. Integrate the first order system over $[x_0, 0.1]$ with default error tolerances. Plot the solution that you compute along with an approximate solution obtained from the series.

Bibliography

[1] P. Amodio, J.R. Cash, G. Roussos, R.W. Wright, G. Fairweather, I. Gladwell, G.L. Kraut and M. Paprzycki, Almost block diagonal linear systems: sequential and parallel solution techniques, and applications, *Numerical Linear Algebra and its Applications*, **7** (2000), pp. 275–317.

[2] W. Ames W. and E. Lohner, Nonlinear models of reaction–diffusion in rivers, pp. 217–219 in R. Vichnevetsky and R. Stepleman, eds., *Advances in Computer Methods for Partial Differential Eqautions–IV*, IMACS, New Brunswick, NJ, 1981.

[3] D. Arnold and J.C. Polking, *Ordinary Differential Equations Using* MATLAB (ed. 2), Prentice–Hall,Englewood Cliffs, NJ, 1999.

[4] U.M. Ascher, J. Christiansen and R.D. Russell, COLSYS – a collocation code for boundary value problems, pp. 164–185 in B. Childs et alia, eds., *Codes for Boundary Value Problems*, Springer Lecture Notes in Comp. Sc., **76**, Springer–Verlag, New York, NY, 1979.

[5] U.M. Ascher, J. Christiansen and R.D. Russell. Collocation software for boundary value ODE's, *ACM Trans. Math. Soft.*, **7** (1981), pp. 209–229.

[6] U.M. Ascher and R.D. Russell, Reformulation of boundary value problems into 'standard' form, *SIAM Review*, **23** (1981), pp. 238–254.

[7] U.M. Ascher, R.M.M. Mattheij and R.D. Russell, *Numerical Solution of Boundary Value Problems for Ordinary Differential Equations*, SIAM, Philadelphia, 1995.

[8] G. Bader and U. Ascher. A new basis implementation for a mixed order boundary value solver. *SIAM J. Sci. Stat. Comp.*, **9** (1987), pp. 483–500.

[9] P.B. Bailey, B.S. Garbow, H.G. Kaper and A. Zettl, Eigenvalue and eigenfunction computations for Sturm–Liouville problems, *ACM Trans. Math. Softw.*, **17** (1991), pp. 491–499.

[10] P.B. Bailey, M.K. Gordon and L.F. Shampine, Automatic solution of the Sturm–Liouville problem, *ACM Trans. Math. Softw.*, **4** (1978), pp. 193–208.

[11] P.B. Bailey, L.F. Shampine and P.E. Waltman, *Nonlinear Two Point Boundary Value Problems*, Academic Press, New York, 1968.

[12] C.T.H. Baker, C.A.H. Paul, and D.R. Willé, *A bibliography on the numerical solution of delay differential equations*, Numerical Analysis Report **269**, Mathematics Department, University of Manchester, U.K., 1995.

[13] C.T.H. Baker, C.A.H. Paul, and D.R. Willé, Issues in the numerical solution of evolutionary delay differential equations, *Adv. Comput. Math.*, **3** (1995), pp. 171–196.

[14] C.M. Bender and S.A. Orszag, *Advanced Mathematical Methods for Scientists and Engineers I, Asymptotic Methods and Perturbation Theory*, Springer–Verlag, New York, NY, 1999.

[15] R.W. Brankin, J.R. Dormand, I. Gladwell, P. Prince and W.L. Seward, ALGORITHM 670: A Runge–Kutta–Nyström code, *ACM Trans. Math. Softw.*, **15** (1989), pp. 31–40.

[16] R.W. Brankin and I. Gladwell, A Fortran 90 version of RKSUITE: An ODE initial value solver, *Ann. Numer. Math.*, **1** (1994), pp. 363–375.

[17] R.W. Brankin, I. Gladwell and L.F. Shampine, RKSUITE: A suite of explicit Rünge–Kutta codes, pp. 41–53 in R.P. Agarwal, ed., *Contributions to Numerical Mathematics*, *WSSIAA* **2**, World Scientific Press, Singapore, 1993.

[18] K.E. Brenan, S.L. Campbell and L.R. Petzold, *Numerical Solution of Initial–Value Problems in Differential–Algebraic Equations*, SIAM Classics in Applied Mathematics **14**, SIAM, Philadelphia, PA, 1996.

[19] P.N. Brown, G.D. Byrne, and A.C. Hindmarsh, VODE: a variable coefficient ODE solver, *SIAM J. Sci. Stat. Comput.*, **10** (1989), pp. 1038–1051.

[20] P. Bogacki and L.F. Shampine, A 3(2) pair of Runge–Kutta formulas, *Appl. Math. Letters*, **2** (1989), pp. 1–9.

[21] R.L. Borrelli and C.S. Coleman, *ODE Architect*, J. Wiley & Sons, New York, NY, 1999.

[22] J.R. Cash and M.H. Wright, A deferred correction method for nonlinear two–point boundary value problems: Implementation and numerical evaluation, *SIAM J. Sci. Stat. Comput.*, **12** (1991), pp. 971–989.

[23] T.K. Caughy, Large amplitude whirling of an elastic string—a nonlinear eigenvalue problem, *SIAM J. Appl. Math.*, **18** (1970), pp. 210–237.

[24] T. Cebeci and H.B. Keller, Shooting and parallel shooting methods for solving the Falkner–Skan boundary–layer equations, *J. Comp. Physics*, **7** (1971), pp. 289–300.

[25] J.D. Cole, *Perturbation Methods in Applied Mathematics*, Blaisdell, Waltham, MA, 1968.

[26] K. Cooke, P. van den Driessche, and X. Zou, Interaction of maturation delay and nonlinear birth in population and epidemic models, *J. Math. Biol.*, **39** (1999), pp. 332–352.

[27] S.P. Corwin, D. Sarafyan, and S. Thompson, DKLAG6: a code based on continuously imbedded sixth order Runge–Kutta methods for the solution of state dependent functional differential equations, *Appl. Numer. Math.*, **24** (1997), pp. 319–333.

[28] S.P. Corwin and S. Thompson, *DKLAG6: solution of systems of functional differential equations with state dependent delays*, **TR–96–002**, Computer Science Department Technical Report Series, Radford University, Radford, VA, 1996.

[29] S.P Corwin and S. Thompson, *DRAKE: continuous simulation software for the solution of delay differential equations on personal computers*, **TR–93–001**, Computer Science Department Technical Report Series, Radford University, Radford, VA, 1993.

[30] A.R. Curtis, M.J.D. Powell and J.K. Reid, On the estimation of sparse Jacobian matrices, *J. Inst. Math. Appl.*, **13** (1974), pp. 117–119.

[31] H.T. Davis, *Introduction to Nonlinear Differential and Integral Equations*, Dover, New York, NY, 1962.

[32] F.R. de Hoog and R. Weiss, Difference methods for boundary value problems with a singularity of the first kind, *SIAM J. Numer. Anal.* **13** (1976), 775–813.

[33] F.R. de Hoog and R. Weiss, Collocation methods for singular boundary value problems, *SIAM J. Numer. Anal.* **15** (1978), 198–217.

[34] J.R. Dormand, *Numerical Methods for Differential Equations*, CRC Press, Boca Raton, FL, 1996.

[35] J.R. Dormand and P.J. Prince, A family of embedded Runge–Kutta formulae, *J. Comput. Appl. Math.*, **27** (1980), pp. 19–26.

[36] C.H. Edwards, Newton's nose–cone problem, *The Mathematica Journal*, **7** (1997), pp. 64–71.

[37] W.H. Enright and H. Hayashi, A delay differential equation solver based on a continuous Runge–Kutta method with defect control, *Numer. Alg.*, **16** (1997), pp. 349–364.

[38] W.H. Enright and H. Hayashi, Convergence analysis of the solution of retarded and neutral differential equations by continuous methods, *SIAM J. Numer. Anal.*, **35** (1998), pp. 572–585.

[39] W.H. Enright and P.H. Muir, Runge–Kutta software with defect control for boundary value ODEs, *SIAM J. Sci. Comput.*, **17** (1996), pp. 479–497.

[40] J.D. Farmer, Chaotic attractors of an infinite–dimensional dynamical system, *Physica D*, **4** (1982), pp. 366–393.

[41] E. Fehlberg, Klassiche Runge-Kutta-Formeln vierter und niedrigerer Ordnung mit Schrittenweiten–Kontrolle und ihre Anwendung auf Wärmeleitungsprobleme, *Computing*, **6** (1970), pp. 61–71.

[42] B.A. Finlayson, *The Method of Weighted Residuals and Variational Principles*, Academic Press, New York, NY, 1972.

[43] C.A.J. Fletcher, *Computational Galerkin Methods*, Springer–Verlag, New York, NY, 1984.

[44] *GAMS, the Guide to Available Mathematical Software* is available at http://gams.nist.gov/.

[45] C.W. Gear, *Numerical Initial Value Problems in Ordinary Differential Equations*, Prentice–Hall, Englewood Cliffs, NJ, 1971.

[46] L. Genik and P. van den Driessche, An epidemic model with recruitment–death demographics and discrete delays, pp. 237–249 in S. Ruan, G.S.K. Wolkowicz and J. Wu, eds., *Differential Equations with Applications to Biology*, American Mathematical Society, Providence, RI, 1999.

[47] F.R. Giordano and M.D. Weir, *Differential Equations: A Modeling Approach*, Addison–Wesley, Reading, MA, 1991.

[48] I. Gladwell, The development of the boundary–value codes in the ordinary differential equations chapter of the NAG library, pp. 122–143 in B. Childs et alia, eds., *Codes for Boundary Value Problems*, Springer Lecture Notes in Computer Science, **76**, Springer–Verlag, New York, NY, 1979.

[49] I. Gladwell, Initial value routines in the NAG library, *ACM Trans. Math. Softw.*, **5** (1979), pp. 386–400.

[50] I. Gladwell, *The NAG library boundary value codes*, Numerical Analysis Report **134**, Department of Mathematics, University of Manchester, U.K., 1987.

[51] E. Hairer, S.P. Nörsett, and G. Wanner, *Solving Ordinary Differential Equations I*, Springer–Verlag, Berlin, Germany, 1987.

[52] E. Hairer and G. Wanner, *Solving Ordinary Differential Equations II, Stiff and Differential–Algebraic Problems*, Springer–Verlag, Berlin, Germany, 1991.

[53] J. Hale, *Functional Differential Equations*, Springer–Verlag, Berlin, Germany, 1971.

[54] A.H. Harvey, A.P. Peskin and S.A. Klein, *NIST Standard Reference Database 10*, Version **2.2**, National Institutes of Standards and Technology, Gaithersburg, MD, 1996.

[55] The *Harwell 2000 Library*, at hsl.rl.ac.uk.

[56] P. Henrici, *Discrete Variable Methods in Ordinary Differential Equations*, J. Wiley & Sons, New York, NY, 1962.

[57] P. Henrici, *Error Propagation for Difference Methods*, Krieger, New York, NY, 1977.

[58] D.J. Higham and N.J. Higham, MATLAB *Guide*, SIAM, Philadelphia, PA, 2000.

[59] A.C. Hindmarsh and G.D. Byrne, Applications of EPISODE: an experimental package for the integration of systems of ordinary differential equations, pp. 147–166 in L. Lapidus and W.E. Schiesser, eds., *Numerical Methods for Differential Systems*, Academic Press, New York, NY, 1976.

[60] M.H. Holmes, *Introduction to Perturbation Methods*, Springer–Verlag, New York, NY, 1995.

[61] W. Huang, Y. Ren, and R.D. Russell, Moving mesh methods based on moving mesh partial differential equations, *J. Comput. Phys.*, **113** (1994), pp. 279–290.

[62] T.E. Hull, W.H. Enright, B.M. Fellen, and A.E. Sedgwick, Comparing numerical methods for ordinary differential equations, *SIAM J. Numer. Anal.*, **9** (1972), pp. 603–637.

[63] T.E. Hull, W.H. Enright, and K.R. Jackson, *User's guide for DVERK—a subroutine for solving non–stiff ODEs*, Report **100**, Computer Science Department, University of Toronto, Canada, 1975.

[64] *The IMSL FORTRAN 77 Mathematics and Statistics Libraries (FNL)*, Version **3.0**, Visual Numerics Inc., Houston, TX, 2002.

[65] E. Isaacson and H.B. Keller, *Analysis of Numerical Methods*, J. Wiley & Sons, New York, NY, 1966.

[66] E. Kamke, *Differentialgleichungen Lösungsmethoden und Lösungen*, vol. I, Chelsea, New York, NY, 1971.

[67] H.B. Keller, *Numerical Methods for Two–Point Boundary–Value Problems*, Dover, New York, NY, 1992.

[68] J. Kierzenka, *Studies in the Numerical Solution of Ordinary Differential Equations*, Doctoral Dissertation, Department of Mathematics, Southern Methodist University, Dallas, TX, 1998.

[69] J. Kierzenka and L.F. Shampine, A BVP solver based on residual control and the MATLAB PSE, *ACM Trans. Math. Softw.*, **27** (2001), pp.299–316.

[70] H. Koçak, *Differential and Difference Equations through Computer Experiments*, Springer–Verlag, New York, NY, 1989.

[71] M. Kubiček, V. Hlaváček and M. Holodnick, Test examples for comparison of codes for nonlinear boundary value problems in ordinary differential equations, pp. 325–346 in B. Childs et alia, eds., *Codes for Boundary–Value Problems in Ordinary Differential Equations*, Lecture Notes in Computer Science, **76**, Springer–Verlag, New York, NY, 1979.

[72] J.D. Lambert, *Numerical Methods for Ordinary Differential Systems*, J. Wiley & Sons, New York, NY, 1991.

[73] L. Lapidus, R.C. Aiken, and Y.A. Liu, The occurrence and numerical solution of physical and chemical systems having widely varying time constants, pp. 187–200 in R.A. Willoughby, ed., *Stiff Differential Systems*, Plenum Press, New York, NY, 1973.

[74] H.T. Laquer and B. Wendroff, Bounds for the model quench front, *SIAM J. Numer. Anal.*, **18** (1981), pp. 225–241.

[75] M. Lentini and V. Pereyra, A variable order finite difference method for nonlinear multipoint boundary value problems, *Math. Comp.*, **23** (1974), pp. 981–1003.

[76] J. Lighthill, *An Informal Introduction to Theoretical Fluid Mechanics*, Clarendon Press, Oxford, 1986.

[77] C.C. Lin and L.A. Segel, *Mathematics Applied to Deterministic Problems in the Natural Sciences*, SIAM, Philadelphia, PA, 1988.

[78] N. MacDonald, *Time Lags in Biological Models*, Springer–Verlag, Berlin, Germany, 1978.

[79] N. MacDonald, *Biological Delay Systems: Linear Stability Theory*, Cambridge University Press, Cambridge, U.K., 1989.

[80] *Maple V Release 6*, Waterloo Maple Inc., 450 Phillip St., Waterloo, ON, Canada N2L 5J2, 1998.

[81] M. Marletta and J.D. Pryce, LCNO Sturm–Liouville problems—Computational difficulties and examples, *Numer. Math.*, **69** (1995), pp.303–320.

[82] C. Marriott and C. DeLisle, Effects of discontinuities in the behavior of a delay differential equation, *Physica D*, **36** (1989), pp. 198–206.

[83] A. Martin and S. Ruan, Predator–prey models with delay and prey harvesting, *J. Math. Biol.*, **43** (2001), pp. 247–267.

[84] MATLAB *6*, The MathWorks, Inc., 3 Apple Hill Dr., Natick, MA 01760, 2000.

[85] R.M.M. Mattheij and G.W.M. Staarink, An efficient algorithm for solving general linear two–point BVP, *SIAM J. Sci. Stat. Comp.*, **5** (1984), pp. 745–763.

[86] R.M.M. Mattheij and G.W.M. Staarink, On optimal shooting intervals, *Math. Comp.*, **42** (1984), pp. 25–40.

[87] C.B. Moler, Are we there yet?, MATLAB *Newsletter*, Simulink 2 Special Edition, 1997, pp. 16–17. See http://www.mathworks.com/company/newsletter/pdf/97slCleve.pdf.

[88] C.B. Moler and L.P. Solomon, Integrating square roots, *Comm. ACM*, **13** (1970), pp. 556–557.

[89] J.S. Murphy, Extensions of the Falkner–Skan similar solutions to flows with surface curvature, *AIAA J.*, **3** (1965), pp. 2043–2049.

[90] J.D. Murray, *Mathematical Biology*, 2nd ed., Springer–Verlag, Berlin, Germany, 1993.

[91] *NAG FORTRAN 77 Library*, **Mark 21**, Numerical Algorithms Group Inc., Oxford, U.K., 2002.

[92] The *Netlib* software repository is available at http://www.netlib.org/.

[93] K.W. Neves, Automatic integration of functional differential equations: an approach, *ACM Trans. Math. Softw.*, **1** (1975), pp. 357–368.

[94] K.W. Neves and S. Thompson, Software for the numerical solution of systems of functional differential equations with state dependent delays, *Appl. Numer. Math.*, **9** (1992), pp. 385–401.

[95] H.J. Oberle and H.J. Pesch, Numerical treatment of delay differential equations by Hermite interpolation, *Numer. Math.*, **37** (1981), pp. 235–255.

[96] R.E. O'Malley, *Singular Perturbation Methods for Ordinary Differential Equations*, Springer–Verlag, New York, NY, 1991.

[97] J.M. Ortega and W.G. Poole, *An Introduction to Numerical Methods for Differential Equations*, Pitman, Marshfield, MA, 1981.

[98] J.T. Ottesen, Modelling of the baroflex–feedback mechanism with time–delay, *J. Math. Biol.*, **36** (1997), pp. 41–63.

[99] C.A.H. Paul, *A user–guide to ARCHI*, Numerical Analysis Report **283**, Mathematics Department, University of Manchester, U.K., 1995.

[100] S. Pruess, C.T. Fulton and Y. Xie, *Performance of the Sturm–Liouville software SLEDGE*, Technical Report **MCS–91–19**, Department of Mathematical Sciences, Colorado School of Mines, Revised 1992.

[101] J.D. Pryce, *Numerical Solution of Sturm–Liouville Problems*, Clarendon Press, Oxford, U.K., 1993.

[102] J.D. Pryce, A test package for Sturm–Liouville solvers, *ACM Trans. Math. Softw.*, **25** (1999), pp. 21–57.

[103] A. Raghothama and S. Narayanan, Periodic response and chaos in nonlinear systems with parametric excitation and time delay, *Nonlin. Dyn.*, **27** (2002), pp. 341–365.

[104] S.M. Roberts and J.S. Shipman, *Two–Point Boundary Value Problems: Shooting Methods*, Elsevier, New York, NY, 1972.

[105] H.H. Robertson, The solution of a set of reaction rate equations, pp. 178–182 in J. Walsh, ed., *Numerical Analysis: An Introduction*, Academic Press, London, U.K., 1966.

[106] J.M. Sanz–Serna and M.P. Calvo, *Numerical Hamiltonian Problems*, Chapman & Hall, London, U.K., 1994.

[107] M.R. Scott, *Invariant Imbedding and its Applications to Ordinary Differential Equations, An Introduction*, Addison–Wesley, Reading, Mass., MA, 1973.

[108] R. Seydel, *From Equilibrium to Chaos*, Elsevier, New York, NY, 1988.

[109] L.F. Shampine, Conservation laws and the numerical solution of ODEs, *Comp. & Maths. with Appls.*, **12B** (1986), pp. 1287–1296.

[110] L.F. Shampine, *Numerical Solution of Ordinary Differential Equations*, Chapman & Hall, New York, NY, 1994.

[111] L.F. Shampine, Linear conservation laws for ODEs, *Comp. & Maths. with Appls.*, **35** (1998), pp. 45–53.

[112] L.F. Shampine, Variable order Adams codes, *Comp. & Maths. with Appls.*, to appear.

[113] L.F. Shampine, R.C. Allen, Jr., and S. Pruess, *Fundamentals of Numerical Computing*, J. Wiley & Sons, New York, NY, 1997.

[114] L.F. Shampine, I. Gladwell and R.W. Brankin, Reliable solution of special root finding problems for ODE's, *ACM Trans. Math. Softw.*, **17** (1991), pp. 11–25.

[115] L.F. Shampine and M.K. Gordon, *Computer Solution of Ordinary Differential Equations*, W.H. Freeman, San Francisco, CA, 1975.

[116] L.F. Shampine and M.W. Reichelt, The MATLAB ODE suite, *SIAM J. Sci. Comput.*, **18** (1997), pp. 1–22.

[117] L.F. Shampine, M.W. Reichelt, and J.A. Kierzenka, Solving index–1 DAEs in MATLAB and Simulink, *SIAM Review*, **41** (1999), pp. 538–552.

[118] L.F. Shampine and S. Thompson, Event location for ordinary differential equations, *Comp. & Maths. with Appls.*, **39** (2000), pp. 43–54.

[119] L.F. Shampine and S. Thompson, Solving DDEs in MATLAB, *Appl. Numer. Math.*, **37** (2001), pp. 441–458.

[120] R.D. Skeel and M. Berzins, A method for the spatial discretization of parabolic equations in one space variable, *SIAM J. Sci. Stat. Comput.*, **11** (1990), pp. 1–32.

[121] S.S. Soliman and M.D. Srinath, *Continuous and Discrete Signals and Systems* (ed. 2), Prentice–Hall, Englewood Cliffs, NJ, 1998.

[122] F. Song, Department of Chemical Engineering, University of Toronto, Canada, private communication, 2000.

[123] A.M. Stuart and A.R. Humphries, *Dynamical Systems and Numerical Analysis*, Cambridge University Press, Cambridge, U.K., 1996.

[124] H.J. Stetter, *Analysis of Discretization Methods for Ordinary Differential Equations*, Springer–Verlag, New York, NY, 1973.

[125] S. Suherman, R.H. Plaut, L.T. Watson, and S. Thompson, Effect of human response time on rocking instability of a two–wheeled suitcase, *J. of Sound and Vibration*, **207** (1997), pp. 617–625.

[126] L. Tavernini, *Continuous–Time Modeling and Simulation*, Gordon & Breach, Amsterdam, Netherlands, 1996.

[127] S. Thompson and P.G. Tuttle, Benchmark fluid flow problems for continuous simulation languages, *Comput. & Maths. with Applics.*, **12A** (1986), pp. 345–352.

[128] L.N. Trefethen, *Spectral Methods in* MATLAB, SIAM, Philadelphia, PA, 2000.

[129] *Using* MATLAB, The MathWorks, Inc., 3 Apple Hill Dr., Natick, MA 01760, 1996.

[130] D.A. Wells, *Theory and Problems of Lagrangian Dynamics*, Schaum's Outline Series, McGraw–Hill, New York, NY, 1967.

[131] D.R. Willé and C.T.H. Baker, DELSOL – a numerical code for the solution of systems of delay–differential equations, *Appl. Numer. Math.*, **9** (1992), pp. 223–234.

[132] S.J. Wolfram, *The Mathematica Book*, 3rd ed., Wolfram Media & Cambridge University Press, New York, NY, 1996.