Solving ODEs with MATLAB

This book is for people who need to solve ordinary differential equations (ODEs), both initial value problems (IVPs) and boundary value problems (BVPs) as well as delay differential equations (DDEs). These topics are usually taught in separate courses of length one semester each, but *Solving ODEs with MATLAB* provides a sound treatment of all three in about 250 pages. The chapters on each of these topics begin with a discussion of "the facts of life" for the problem, mainly by means of examples. Numerical methods for the problem are then developed – but only the methods most widely used. Although the treatment of each method is brief and technical issues are minimized, the issues important in practice and for understanding the codes are discussed. Often solving a real problem is much more than just learning how to call a code. The last part of each chapter is a tutorial that shows how to solve problems by means of small but realistic examples.

About the Authors

L. F. Shampine is Clements Professor of Applied Mathematics at Southern Methodist University in Dallas, Texas.

I. Gladwell is Professor of Mathematics at Southern Methodist University in Dallas, Texas.

S. Thompson is Professor of Mathematics at Radford University in Radford, Virginia.

This book distills decades of experience helping people solve ODEs. The authors accumulated this experience in industrial and laboratory settings that include NAG (Numerical Algorithms Group), Babcock and Wilcox Company, Oak Ridge National Laboratory, Sandia National Laboratories, and The MathWorks – as well as in academic settings that include the University of Manchester, Radford University, and Southern Methodist University. The authors have contributed to the subject by publishing hundreds of research papers, writing or editing a half-dozen books, editing leading journals, and writing mathematical software that is in wide use. With associates at The MathWorks, Inc., they wrote all the programs for solving ODEs in Matlab, programs that are the foundation of this book.

published by the press syndicate of the university of cambridge
The Pitt Building, Trumpington Street, Cambridge, United Kingdom

cambridge university press
The Edinburgh Building, Cambridge CB2 2RU, UK
40 West 20th Street, New York, NY 10011-4211, USA
477 Williamstown Road, Port Melbourne, VIC 3207, Australia
Ruiz de Alarcón 13, 28014 Madrid, Spain
Dock House, The Waterfront, Cape Town 8001, South Africa

http://www.cambridge.org

First published 2003

Printed in the United States of America

Typeface Times 10.5/13 pt. *System* AMS-TEX [FH]

A catalog record for this book is available from the British Library.

Library of Congress Cataloging in Publication data available

ISBN 0 521 82404 4 hardback
ISBN 0 521 53094 6 paperback

Contents

Chapter 1

Getting Started

1.1 Introduction

Ordinary differential equations (ODEs) are used throughout engineering, mathematics, and science to describe how physical quantities change, so an introductory course on elementary ODEs and their solutions is a standard part of the curriculum in these fields. Such a course provides insight, but the solution techniques discussed are generally unable to deal with the large, complicated, and nonlinear systems of equations seen in practice. This book is about solving ODEs numerically. Each of the authors has decades of experience in both industry and academia helping people like yourself solve problems. We begin in this chapter with a discussion of what is meant by a numerical solution with standard methods and, in particular, of what you can reasonably expect of standard software. In the chapters that follow, we discuss briefly the most popular methods for important classes of ODE problems. Examples are used throughout to show how to solve realistic problems. Matlab (2000) is used to solve nearly all these problems because it is a very convenient and widely used *problem-solving environment* (PSE) with quality solvers that are exceptionally easy to use. It is also such a high-level programming language that programs are short, making it practical to list complete programs for all the examples. We also include some discussion of software available in other computing environments. Indeed, each of the authors has written ODE solvers widely used in general scientific computing.

An ODE represents a relationship between a function and its derivatives. One such relation taken up early in calculus courses is the linear ordinary differential equation

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

which is to hold for, say, $0 \le t \le 10$. As we learn in a first course, we need more than just an ODE to specify a solution. Often solutions are specified by means of an initial value. For example, there is a unique solution of the ODE (1.1) for which $y(0) = 1$, namely $y(t) = e^t$. This is an example of an *initial value problem* (IVP) for an ODE. Like this example, the IVPs that arise in practice generally have one and only one solution. Sometimes solutions are specified in a more complicated way. This is important in practice, but it is not often discussed in a first course except possibly for the special case of Sturm–Liouville eigenproblems. Suppose that $y(x)$ satisfies the equation

$$y''(x) + y(x) = 0 \tag{1.2}$$

for $0 \le x \le b$. When a solution of this ODE is specified by conditions at both ends of the interval such as

$$y(0) = 0, \quad y(b) = 0$$

we speak of a *boundary value problem* (BVP). A Sturm–Liouville eigenproblem like this BVP always has the trivial solution $y(x) \equiv 0$, but for certain values of b there are nontrivial solutions, too. For instance, when $b = 2\pi$, the BVP has infinitely many solutions of the form $y(x) = \alpha \sin(x)$ for any constant α. In contrast to IVPs, which usually have a unique solution, the BVPs that arise in practice may have no solution, a unique solution, or more than one solution. If there is more than one solution, there may be a finite number or an infinite number of them.

Equation (1.1) tells us that the rate of change of the solution at time t is equal to the value of the solution then. In many physical situations, the effects of changes to the solution are delayed until a later time. Models of this behavior lead to *delay differential equations* (DDEs). Often the delays are taken to be constant. For example, if the situation modeled by the ODE (1.1) is such that the effect of a change in the solution is delayed by one time unit, then the DDE is

$$y'(t) = y(t-1) \tag{1.3}$$

for, say, $0 \le t \le 10$. This problem resembles an initial value problem for an ODE; when the delays are constant, both the theory of DDEs and their numerical solution can be based on corresponding results for ODEs. There are, however, important differences. For the ODE (1.1), the initial value $y(0) = 1$ is enough to determine the solution, but that cannot be enough for the DDE (1.3). After all, when $t = 0$ we need $y(-1)$ to define $y'(0)$, but this is a value of the solution prior to the initial time. Thus, an initial value problem for the DDE (1.3) involves not just the value of the solution at the starting time but also its *history*. For this example it is easy enough to argue that, if we specify $y(t)$ for $-1 \le t \le 0$, then the initial value problem has a unique solution.

This book is about solving initial value problems for ODEs, boundary value problems for ODEs, and initial value problems for a class of DDEs with constant delays. For brevity we refer throughout to these three kinds of problems as IVPs, BVPs, and DDEs. In the rest of this chapter we discuss fundamental issues that are common to all three. Indeed, some are so fundamental that – even if all you want is a little help solving a specific problem – you need to understand them. The IVPs are taken up in Chapter 2, BVPs in Chapter 3, and DDEs in Chapter 4. The IVP chapter comes first because the ideas and the software of that chapter are used later in the book, so some understanding of this material is needed to appreciate the chapters that follow. The chapters on BVPs and DDEs are mutually independent.

It is assumed that you are acquainted with the elements of programming in Matlab, so we discuss only matters connected with solving ODEs. If you need to supplement your understanding of the language, the PSE itself has good documentation and there are a number of books available that provide more detail. One that we particularly like is the *MATLAB Guide* (Higham & Higham 2000). Most of the programs supplied with *Solving ODEs with MATLAB* plot solutions on the screen in color. Because it was not practical to provide color figures in the book, we modified the output of these programs to show the solutions in monochrome. Version 6.5 (Release 13) of Matlab is required for Chapter 4, but version 6.1 suffices for the other chapters. Much of the cited software for general scientific computing is available from general-purpose, scientific computing libraries such as NAG (2002), Visual Numerics (IMSL 2002), and Harwell 2000 (*H2KL*), or from the Netlib Repository (*Netlib*). If the source of the software is not immediately obvious, it can be located through the classification system GAMS, the Guide to Available Mathematical Software (*GAMS*).

Numerical methods and the analytical tools of classical applied mathematics are complementary techniques for investigating and undertaking the solution of mathematical problems. You might be able to solve analytically simple equations involving a very few unknowns, especially with the assistance of a PSE for computer algebra like Maple (1998) or Mathematica (Wolfram 1996). All our examples were computed using the Maple kernel provided with the student version of Matlab or using the Symbolic Toolbox provided with the professional version.

First we observe that even small changes to the equations can complicate greatly the analytical solutions. For example, Maple is used via Matlab to solve the ODE

$$y' = y^2$$

at the command line by

```
>> y = dsolve('Dy = y^2')

y = -1/(t-C1)
```

(Sometimes we edit output slightly to give a more compact display.) In this general solution `C1` is an arbitrary constant. This family of solutions expressed in terms of a familiar function gives us a lot of insight about how solutions behave. If the ODE is changed "slightly" to

$$y' = y^2 + 1$$

then the general solution is found by `dsolve` to be

```
  y = tan(t+C1)
```

This is more complicated because it expresses the solution in terms of a special function, but it is at least a familiar special function and we understand well how it behaves. However, if the ODE is changed to

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

then the general solution found by `dsolve` is

```
  y = (C1*AiryAi(1,-t)+AiryBi(1,-t))/
      (C1*AiryAi(-t)+AiryBi(-t))
```

which in standard mathematical notation is

$$y(t) = \frac{C_1 Ai'(-t) + Bi'(-t)}{C_1 Ai(-t) + Bi(-t)}$$

Here $Ai(t)$ and $Bi(t)$ are Airy functions. (The Maple kernel denotes these functions by `AiryAi` and `AiryBi`, cf. `mhelp airy`; but `Matlab` itself uses different names, cf. `help airy`.) Again `C1` is an arbitrary constant. The Airy functions are not so familiar. This solution is useful for studying the behavior of solutions analytically, but we'd need to plot some solutions to gain a sense of how they behave. Changing the ODE to

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

changes the general solution found by `dsolve` to

```
  y = -t*(C1*besselj(-3/4,1/2*t^2)+bessely(-3/4,1/2*t^2))/
         (C1*besselj(1/4,1/2*t^2)+bessely(1/4,1/2*t^2))
```

which in standard mathematical notation is

$$y(t) = -t\,\frac{C_1 J_{-3/4}\left(\dfrac{t^2}{2}\right) + Y_{-3/4}\left(\dfrac{t^2}{2}\right)}{C_1 J_{1/4}\left(\dfrac{t^2}{2}\right) + Y_{1/4}\left(\dfrac{t^2}{2}\right)}$$

Again the solution is expressed in terms of special functions, but now they are Bessel functions of fractional order. Again, we'd need to plot some solutions to gain insight. These equations are taken up later in Example 2.3.1.

Something different happens if we change the power of y:

```
>> y = dsolve('Dy = y^3 + t^2')
Warning: Explicit solution could not be found.
```

This example shows that even simple-looking equations may not have a solution $y(t)$ that can be expressed in terms of familiar functions by Maple. Such examples are not rare, and usually when Maple fails to find an explicit solution it is because none is known. In fact, for a system of ODEs it is rare that an explicit solution can be found.

For these scalar ODEs it was easy to use a computer algebra package to obtain analytical solutions. Let us now consider some of the differences between solving ODEs analytically and numerically. The analytical solutions of the examples provide valuable insight, but to understand them better we'd need to evaluate and plot some particular solutions. For this we'd need to turn to numerical schemes for evaluating the special functions. But if we must use numerical methods for this, why bother solving them analytically at all? A direct numerical solution might be the best way to proceed for a particular IVP, but Airy and Bessel functions incorporate behavior that can be difficult for numerical methods to reproduce – namely, some have singularities and some oscillate very rapidly. If this is true of the solution that interests us or if we are interested in the solution as $t \to \infty$, then we may not be able to compute the solution numerically in a straightforward way. In effect, the analytical solution isolates the difficulties and we then rely upon the quality of the software for evaluating the special functions to compute an accurate solution. As the examples show, small changes to the ODE can lead to significant changes in the form of the analytical solution, though this may not imply that the behavior of the solution itself changes much. In contrast, there is really no difference solving IVPs numerically for these equations, including the one for which `dsolve` did not produce a solution. This illustrates the most important virtue of numerical methods: they make it easy to solve a large class of problems. Indeed, our considerable experience is that if an IVP arises in a practical situation, most likely you will not be able to solve it analytically yet you will be able to solve it numerically. On the other hand, the analytical solutions of the examples show how they depend on an arbitrary constant `C1`. Because numerical methods solve one problem at a time, it is not easy to determine how solutions depend on parameters. Such insight can be obtained by combining numerical methods with analytical tools such as variational equations and perturbation methods. Another difference between analytical and numerical solutions is that the standard numerical methods of this book apply only to ODEs defined by smooth functions that are to be solved on a finite interval. It is not unusual for physical problems to involve singular points or an infinite interval. Asymptotic expansions are often combined with numerical methods to deal with these difficulties.

In our view, analytical and numerical methods are complementary approaches to solving ODEs. This book is about numerical methods because they are easy to use and broadly applicable, but some kinds of difficulties can be resolved or understood only by analytical means. As a consequence, the chapters that follow feature many examples of using applied mathematics (e.g., asymptotic expansions and perturbation methods) to assist in the numerical solution of ODEs.

1.2 Existence, Uniqueness, and Well-Posedness

From the title of this section you might imagine that this is just another example of mathematicians being fussy. But it is not: it is about whether you will be able to solve a problem at all and, if you can, how well. In this book we'll see a good many examples of physical problems that do not have solutions for certain values of parameters. We'll also see physical problems that have more than one solution. Clearly we'll have trouble computing a solution that does not exist, and if there is more than one solution then we'll have trouble computing the "right" one. Although there are mathematical results that guarantee a problem has a solution and only one, there is no substitute for an understanding of the phenomena being modeled.

Existence and uniqueness are much simpler for IVPs than BVPs, and the class of DDEs we consider can be understood in terms of IVPs, so we concentrate here on IVPs and defer to later chapters a fuller discussion of BVPs and DDEs. The vast majority of IVPs that arise in practice can be written as a system of d explicit first-order ODEs:

$$\begin{aligned}
y_1'(t) &= f_1(t, y_1(t), y_2(t), \dots, y_d(t)) \\
y_2'(t) &= f_2(t, y_1(t), y_2(t), \dots, y_d(t)) \\
&\vdots \\
y_d'(t) &= f_n(t, y_1(t), y_2(t), \dots, y_d(t))
\end{aligned}$$

For brevity we generally write this system in terms of the (column) vectors

$$y(t) = \begin{pmatrix} y_1(t) \\ y_2(t) \\ \vdots \\ y_d(t) \end{pmatrix}, \quad f(t, y(t)) = \begin{pmatrix} f_1(t, y(t)) \\ f_2(t, y(t)) \\ \vdots \\ f_d(t, y(t)) \end{pmatrix}$$

as

$$y'(t) = f(t, y(t)) \tag{1.4}$$

An IVP is specified by giving values of all the solution components at an initial point,

$$y_1(a) = A_1,\ y_2(a) = A_2,\ \dots,\ y_d(a) = A_d$$

or, in vector notation,

$$y(a) = A = \begin{pmatrix} A_1 \\ A_2 \\ \vdots \\ A_d \end{pmatrix} \tag{1.5}$$

Using vectors, a system of first-order equations resembles a single equation; in fact, the theory is much the same. However, writing problems as first-order systems is not only convenient for the theory, it is critically important in practice. We'll explain this later and show how to do it.

Roughly speaking, if the function $f(t, y)$ is smooth for all values (t, y) in a region R that contains the initial data (a, A), then the IVP comprising the ODE (1.4) and the initial condition (1.5) has a solution and only one. This settles the existence and uniqueness question for most of the IVPs that arise in practice, but we need to expand on the issue of where the solution exists. The solution extends to the boundary of the region R, but that is not the same as saying that it exists throughout a given interval $a \le t \le b$ contained in the region R. An example makes the point. The IVP

$$y' = y^2, \quad y(0) = 1$$

has a function $f(t, y) = y^2$ that is smooth everywhere; in other words, it is smooth in the region

$$R = \{-\infty < t < \infty,\ -\infty < y < \infty\}$$

Yet the unique solution

$$y(t) = \frac{1}{1 - t}$$

"blows up" as $t \to 1$ and hence does not exist on a whole interval $0 \le t \le 2$ (say) that is entirely contained in R. This does not contradict the existence result because as $t \to 1$, the solution approaches the boundary of the region R in the y variable, a boundary that happens to be at infinity. This kind of behavior is not at all unusual for physical problems. Correspondingly, it is usually reasonable to ask that a numerical scheme approximate a solution well until it becomes too large for the arithmetic of the computer used. Exercises 1.2 and 1.3 take up similar cases.

The form of the ODEs (1.4) and the initial condition (1.5) is standard for IVPs, and in Section 1.3 we look at some examples showing how to write problems in this form. Existence and uniqueness is relatively simple for this standard explicit form, but the properties are more difficult to analyze for equations in the implicit form

$$F(t, y(t), y'(t)) = 0$$

Very simple examples show that both existence and uniqueness are problematic for such equations. For instance, the equation

$$(y'(t))^2 + 1 = 0$$

obviously has no (real) solutions. A more substantial example helps make the point. In scientific and engineering applications there is a great deal of interest in how the solutions y of a system of algebraic equations

$$F(y, \lambda) = 0$$

depend on a (scalar) parameter λ. Differentiating with respect to the parameter, we find that

$$\frac{\partial F}{\partial y}\frac{dy}{d\lambda} + \frac{\partial F}{\partial \lambda} = 0$$

This is a system of first-order ODEs. If for some λ_0 we can solve the algebraic equations $F(y, \lambda_0) = 0$ for $y(\lambda_0) = y_0$, then this provides an initial condition for an IVP for $y(\lambda)$. If the Jacobian matrix

$$J = \frac{\partial F}{\partial y} = \left(\frac{\partial F_i}{\partial y_j}\right)$$

is nonsingular, we can write the ODEs in the standard form

$$\frac{dy}{d\lambda} = -J^{-1}\frac{\partial F}{\partial \lambda}$$

However, if the Jacobian matrix is singular then the questions of existence and uniqueness are much more difficult to answer. This is a rather special situation, but in fact it is often the situation with the most interesting science. It is when solutions bifurcate – that is, the number of solutions changes. If we are to apply standard codes for IVPs at such a singular (bifurcation) point, we must resort to the analytical tools of applied mathematics to sort out the behavior of solutions near this point. Exercise 1.1 considers a similar problem.

As a concrete example of bifurcation, suppose that we are interested in steady-state (constant) solutions of the ODE

$$y' = y^2 - \lambda$$

The steady states are solutions of the algebraic equation

$$0 = y^2 - \lambda \equiv F(y, \lambda)$$

It is obvious that, for $\lambda \geq 0$, one steady-state solution is $y(\lambda) = \sqrt{\lambda}$. However, to study more generally how the steady state depends on λ, we could compute it as the solution of the IVP

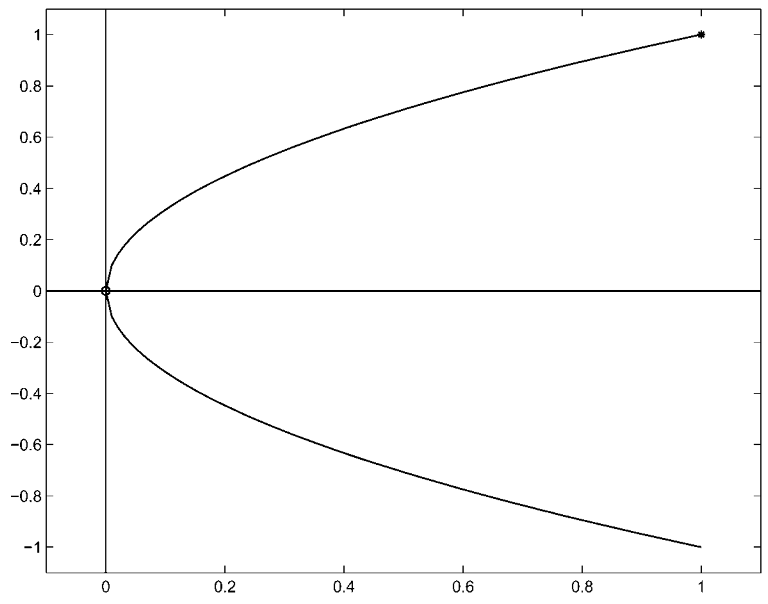

Figure 1.1: $(0, 0)$ is a singular point for $2yy' - 1 = 0$.

$$2y\frac{dy}{d\lambda} - 1 = 0, \quad y(1) = 1$$

Provided that $y \neq 0$, the ODE can be written immediately in standard form and solved for values of λ decreasing from 1. However, the equation is singular when $y(\lambda) = 0$, which is true for $\lambda = 0$. The singular point $(0, 0)$ leaves open the possibility that there is more than one solution of the ODE passing through this point, and so there is: $y(\lambda) = -\sqrt{\lambda}$ is a second solution. Using standard software, we can start at $\lambda = 1$ and integrate the equation easily until close to the origin, where we run into trouble because $y'(\lambda) \to \infty$ as $\lambda \to 0$. See Figure 1.1.

For later use in discussing numerical methods, we need to be a little more precise about what we mean by a *smooth* function $f(t, y)$. We mean that it is continuous in a region R and that it has continuous derivatives with respect to the dependent variables there – as many derivatives as necessary for whatever argument we make. A technical condition is that f must satisfy a *Lipschitz condition* in the region R. That is, there is a constant L such that, for any points (t, u) and (t, v) in the region R,

$$\|f(t, u) - f(t, v)\| \leq L\|u - v\|$$

In the case of a single equation, the mean value theorem states that

$$f(t,u) - f(t,v) = \frac{\partial f}{\partial y}(t,\zeta)(u-v)$$

so $f(t,y)$ satisfies a Lipschitz condition if $\left|\frac{\partial f(t,y)}{\partial y}\right|$ is bounded in the region R by a constant L. Similarly, if the first partial derivatives $\left|\frac{\partial f_i(t,y_1,y_2,\ldots,y_d)}{\partial y_j}\right|$ are all bounded in the region R, then the vector function $f(t,y)$ satisfies a Lipschitz condition there.

Roughly speaking, a *well-posed problem* is one for which small changes to the data lead to small changes in the solution. Such a problem is also said to be *well-conditioned* with respect to changes in the data. This is a fundamental property of a physical problem and it is also fundamental to the numerical solution of the problem. The methods that we study can be regarded as producing the exact solution to a problem with the data that defines the problem changed a little. For a well-posed problem, this means that the numerical solution is close to the solution of the given problem. In practice this is all blurred because it depends both on how much accuracy you want in a solution and on the arithmetic you use in computing it. Let's now discuss a familiar example that illuminates some of the issues.

Imagine that we have a pendulum: a light, rigid rod hanging vertically from a frictionless pivot with a heavy weight (the bob) at the free end. With a particular choice of units, the angle $\theta(t)$ that the pendulum makes with the vertical at time t satisfies the ODE

$$\theta'' + \sin(\theta) = 0 \tag{1.6}$$

Suppose that the pendulum is hanging vertically so that the initial angle $\theta(0) = 0$ and that we thump the bob to give it an initial velocity $\theta'(0)$. When the initial velocity is zero, the pendulum does not move at all. If the velocity is nonzero and small enough, the pendulum will swing back and forth. Figure 1.2 shows $\theta(t)$ for several such solutions, namely those with initial velocities $\theta'(0) = -1.9$, 1.5, and 1.9. There is another kind of solution. If we thump the bob hard enough, the pendulum will swing over the top and, with no friction, it will whirl around the pivot forever. This is to say that if the initial velocity $\theta'(0)$ is large enough then $\theta(t)$ will increase forever. The figure shows two such solutions with initial velocities $\theta'(0) = 2.1$ and 2.5. If you think about it, you'll realize that there is a very special solution that occurs as the solutions change from oscillatory to increasing. This solution is the dotted curve in Figure 1.2. Physically, it corresponds to an initial velocity that causes the pendulum to approach and then come to rest vertically and upside down. Clearly this solution is unstable – an arbitrarily small change to the initial velocity gives rise to a solution that is eventually very different. In other words, the IVP for this initial velocity is ill-posed (ill-conditioned) on long time intervals.

Interestingly, we can deduce the initial velocity that results in the unstable solution of (1.6). This is a conservative system, meaning that the *energy*

$$E(t) = 0.5(\theta'(t))^2 - \cos(\theta(t))$$

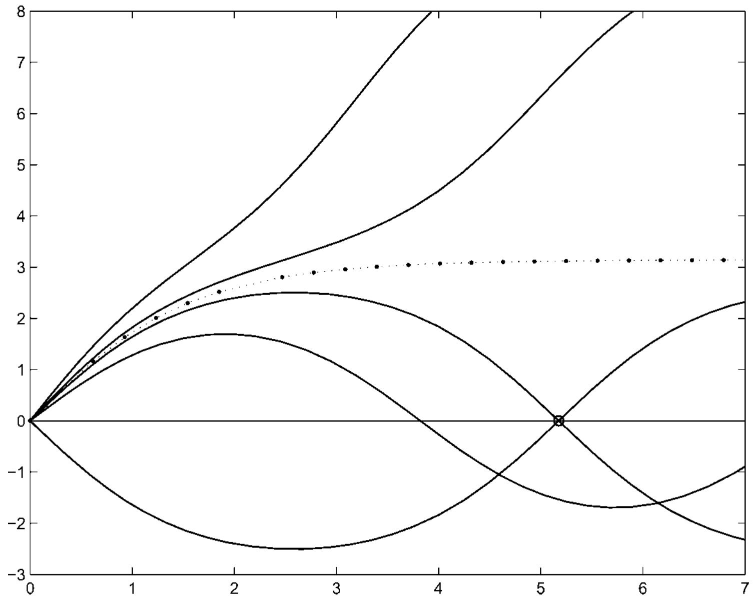

Figure 1.2: $\theta(t)$, the angle from the vertical of the pendulum.

is constant. To prove this, differentiate the expression for $E(t)$ and use the fact that $\theta(t)$ satisfies the ODE (1.6) to see that the derivative $E'(t)$ is zero for all t. On physical grounds, the solution of interest satisfies the condition $\theta(\infty) = \pi$ and, a fortiori, $\theta'(\infty) = 0$. Along with the initial value $\theta(0) = 0$, conservation of energy tells us that for this solution

$$0.5 \times (\theta'(0))^2 - \cos(0) = 0.5 \times 0^2 - \cos(\pi)$$

and hence that $\theta'(0) = 2$. With this we have the unstable solution defined as the solution of the IVP consisting of equation (1.6) and initial values $\theta(0) = 0$ and $\theta'(0) = 2$. The other solutions of Figure 1.2 were computed using the Matlab IVP solver `ode45` and default error tolerances, but these tolerances are not sufficiently stringent to compute an accurate solution of the unstable IVP.

The unstable solution is naturally described as the solution of a boundary value problem. It is the solution of the ODE (1.6) with boundary conditions

$$\theta(0) = 0, \quad \theta(\infty) = \pi \tag{1.7}$$

When modeling a physical situation with a BVP, it is not always clear what boundary conditions to use. We have already commented that, on physical grounds, $\theta'(\infty) = 0$ also. Should we add this boundary condition to (1.7)? No; just as with IVPs, two conditions are needed to specify the solution of a second-order equation and three are too many. But should we use this boundary condition at infinity or should we use $\theta(\infty) = \pi$? A clear difficulty is that, in addition to the solution $\theta(t)$ that we want, the BVP with boundary condition $\theta'(\infty) = 0$ has (at least) two other solutions, namely $-\theta(t)$ and $\theta(t) \equiv 0$. We computed the unstable solution of Figure 1.2 by solving the BVP (1.6) and (1.7) with the Matlab BVP solver `bvp4c`. The solution of the BVP is well-posed, so we could use the default error tolerances. On the other hand, the BVP is posed on an infinite interval, which presents its own difficulties. All the codes we discuss in this book are intended for problems defined on finite intervals. As we see here, it is not unusual for physical problems to be defined on infinite intervals. Existence, uniqueness, and well-posedness are not so clear then. One approach to solving such a problem, which we actually used for the figure, follows the usual physical argument of imposing the conditions at a finite point so distant that it is idealized as being at infinity. For the figure, we solved the ODE subject to the boundary conditions

$$\theta(0) = 0, \quad \theta(100) = \pi$$

It turned out that taking the interval as large as $[0, 100]$ was unnecessarily cautious because the steady state of θ is almost achieved for t as small as 7. For the BVP (1.6) and (1.7), we can use the result $\theta'(0) = 2$ derived earlier as a check on the numerical solution and in particular to check whether the interval is long enough. With default error tolerances, `bvp4c` produces a numerical solution that has an initial slope of $\theta'(0) = 1.999979$, which is certainly good enough for plotting the figure.

Another physical example shows that some BVPs do not have solutions and others have more than one. The equations

$$\begin{aligned} y' &= \tan(\phi) \\ v' &= -\frac{g\sin(\phi) + \nu v^2}{v\cos(\phi)} \\ \phi' &= -\frac{g}{v^2} \end{aligned} \tag{1.8}$$

describe a projectile problem, the planar motion of a shot fired from a cannon. Here the solution component 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 horizontal distance from the cannon. The constant ν represents air resistance (friction) and $g = 0.032$ is the appropriately scaled gravitational constant. These equations neglect three-dimensional effects such as cross winds and rotation of the shot. The initial height is $y(0) = 0$ and there is a given muzzle velocity $v(0)$ for the cannon. The standard projectile problem is to choose the initial angle $\phi(0)$ of the cannon (and hence of the shot) so that the shot will hit a target at the same height as the cannon at distance $x = x_{\text{end}}$. That is, we require $y(x_{\text{end}}) = 0$. All together, the boundary conditions are

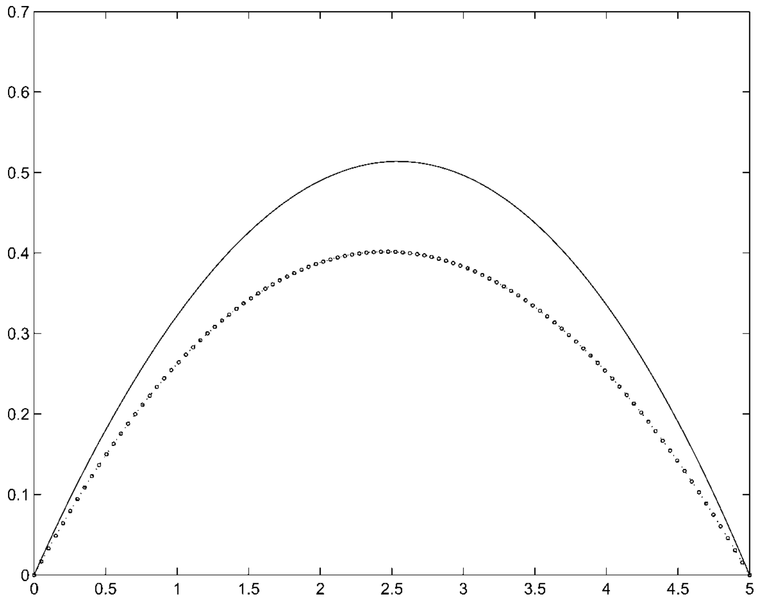

Figure 1.3: Two ways to hit a target at $x_{\text{end}} = 5$ when $v(0) = 0.5$ and $\nu = 0.02$.

$$y(0) = y(x_{\text{end}}) = 0, \quad v(0) \text{ given}$$

Notice that we specify three boundary conditions. Just as with IVPs, for a system of three first-order equations we need three boundary conditions to determine a solution. Does this boundary value problem have a solution? It certainly does not for x_{end} beyond the range of the cannon. On the other hand, if x_{end} is small enough then we expect a solution, but is there only one? No, suppose that the target is close to the cannon. We can hit it by shooting with an almost flat trajectory or by shooting high and dropping the shot on the target. That is, there are (at least) two solutions that correspond to initial angles $\phi(0) = \phi_{\text{low}} \approx 0$ and $\phi(0) = \phi_{\text{high}} \approx \pi/2$. As it turns out, there are exactly two solutions. Now, let x_{end} increase. There are still two solutions, but the larger the value of x_{end}, the smaller the angle ϕ_{high} and the larger the angle ϕ_{low}. Figure 1.3 shows such a pair of trajectories. If we keep increasing x_{end}, eventually we reach the maximum distance possible with the given muzzle velocity. At this distance there is just one solution, $\phi_{\text{low}} = \phi_{\text{high}}$. In summary, there is a critical value of x_{end} for which there is exactly one solution. If x_{end} is smaller than this critical value then there are exactly two solutions; if it is larger, there is no solution at all.

For IVPs we have an existence and uniqueness result that deals with most of the problems that arise physically. There are mathematical results that assert existence and say something about the number of solutions of BVPs, but they are so special that they are seldom important in practice. Instead you must rely on your understanding of the problem to have any confidence that it has a solution and is well-posed. Determining the number of solutions is even more difficult, and in practice about the best we can do is look for a solution close to a guess. There is a real possibility of computing a "wrong" solution or a solution with unexpected behavior.

Stability is the key to understanding numerical methods for the solution of IVPs defined by equation (1.4) and initial values (1.5). All the methods that we study produce approximations $y_n \approx y(t_n)$ on a mesh

$$a = t_0 < t_1 < t_2 < \cdots < t_N = b \tag{1.9}$$

that is chosen by the algorithm. The integration starts with the given initial value $y_0 = y(a) = A$ and, on reaching t_n with $y_n \approx y(t_n)$, the solver computes an approximation at $t_{n+1} = t_n + h_n$. The quantity h_n is called the *step size,* and computing y_{n+1} is described as taking a step from t_n to t_{n+1}.

What the solver does in taking a step is not what you might expect. The *local solution* $u(t)$ is the solution of the IVP

$$u' = f(t,u), \quad u(t_n) = y_n \tag{1.10}$$

In taking a step, the solver tries to find y_{n+1} so that the *local error*

$$u(t_{n+1}) - y_{n+1}$$

is no larger than error tolerances specified by the user. This controls the *true (global) error*

$$y(t_{n+1}) - y_{n+1}$$

only indirectly. The propagation of error can be understood by writing the error at t_{n+1} as

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

The first term on the right is the local error, which is controlled by the solver. The second is the difference at t_{n+1} of two solutions of the ODE that differ by $y(t_n) - y_n$ at t_n. It is a characteristic of the ODE and hence cannot be controlled directly by the numerical method. If the IVP is unstable – meaning that some solutions of the ODEs starting near $y(t)$ spread apart rapidly – then we see from this that the true errors can grow even when the local errors are small at each step. On the other hand, if the IVP is stable so that solutions come together, then the true errors will be comparable to the local errors. Figure 1.2 shows what can happen. As a solver tries to follow the unstable solution plotted with dots, it makes small errors that move the numerical solution on to nearby solution curves. As the figure makes clear, local solutions that start near the unstable solution spread out; the cumulative effect is a very inaccurate numerical solution, even when the solver is able to follow closely each local solution over the span of a single step. It is very important to understand this view of numerical error, for it makes clear a fundamental limitation on all the numerical methods that we consider. No matter how good a job the numerical method does in approximating the solution over the span of a step, if the IVP is unstable then you will eventually compute numerical solutions y_j that are far from the desired solution values $y(t_j)$. How quickly this happens depends on how accurately the method tracks the local solutions and how unstable the IVP is.

A simple example will help us understand the role of stability. The solution of the ODE

$$y' = 5(y - t^2) \tag{1.11}$$

with initial value $y(0) = 0.08$ is

$$y(t) = t^2 + 0.4t + 0.08$$

The IVP and its solution seem innocuous, but the general solution of the ODE is

$$(t^2 + 0.4t + 0.08) + Ce^{5t} \tag{1.12}$$

for an arbitrary constant C. The ODE is unstable because a solution with $C = C_1$ and a solution with $C = C_2$ differ by $(C_1 - C_2)e^{5t}$, a quantity that grows exponentially fast in time. To understand what this means for numerical solution of the IVP, suppose that in the first step we make a small local error so that y_1 is not exactly equal to $y(t_1)$. In the next step we try to approximate the local solution $u(t)$ defined by the ODE and the initial condition $u(t_1) = y_1$. It has the form (1.12) with a small nonzero value of C determined by the initial condition. Suppose that we make no further local errors, so that we compute $y_n = u(t_n)$ for $n = 2, 3, \dots$. The true error then is $y(t_n) - u(t_n) = Ce^{5t_n}$. No matter how small the error in the first step, before long the exponential growth of the true error will result in an unacceptable numerical solution y_n.

For the example of Figure 1.2, the solution curves come together when we integrate from right to left, which is to say that the dotted solution curve is stable in that direction. Sometimes we have a choice of direction of integration, and it is important to appreciate that the stability of IVPs may depend on this direction. The direction field and solution curves for the ODE

$$y' = \cos(t)y \tag{1.13}$$

displayed in Figure 1.4 are illuminating. In portions of the interval, solutions of the ODE spread apart; hence the equation is modestly unstable there. In other portions of the interval, solutions of the ODE come together and the equation is modestly stable. For this equation, the direction of integration is immaterial. This example shows that it is an oversimplification to say simply that an IVP is unstable or stable. Likewise, the growth or decay of errors made at each step by a solver can be complex. In particular, you should

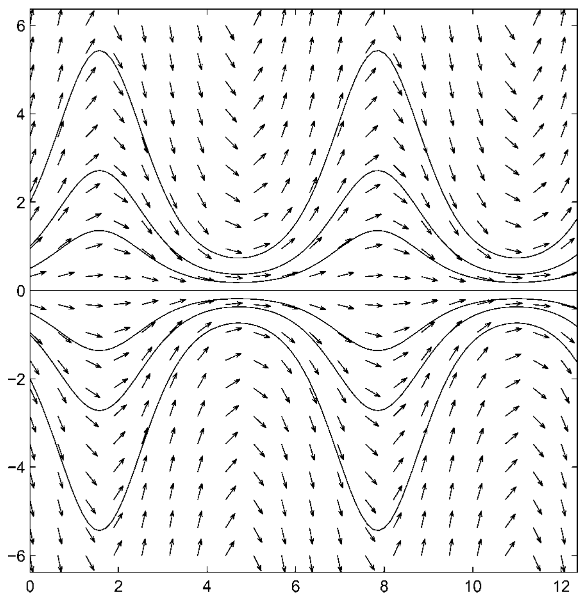

Figure 1.4: Direction field and solutions of the ODE $y' = \cos(t)y$.

not assume that errors always accumulate. For systems of ODEs, one component of the solution can be stable and another unstable at the same time. The coupling of the components of a system can make the overall behavior unclear.

A numerical experiment shows what can happen. Euler's method is a basic scheme discussed fully in the next chapter. It advances the numerical solution of $y' = f(t, y)$ a distance h using the formula

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

The solution of the ODE (1.13) with initial value $y(0) = 2$ is

$$y(t) = 2e^{\sin(t)}$$

The local solution $u(t)$ is the solution of (1.13) that goes through the point (t_n, y_n), namely

$$u(t) = y_n e^{(\sin(t) - \sin(t_n))}$$

Figure 1.5 shows the local and global errors when a constant step size of $h = 0.1$ is used to integrate from $t = 0$ to $t = 3$. Although we are not trying to control the size of the local errors, they do not vary greatly. By definition, the local and global errors are the same in

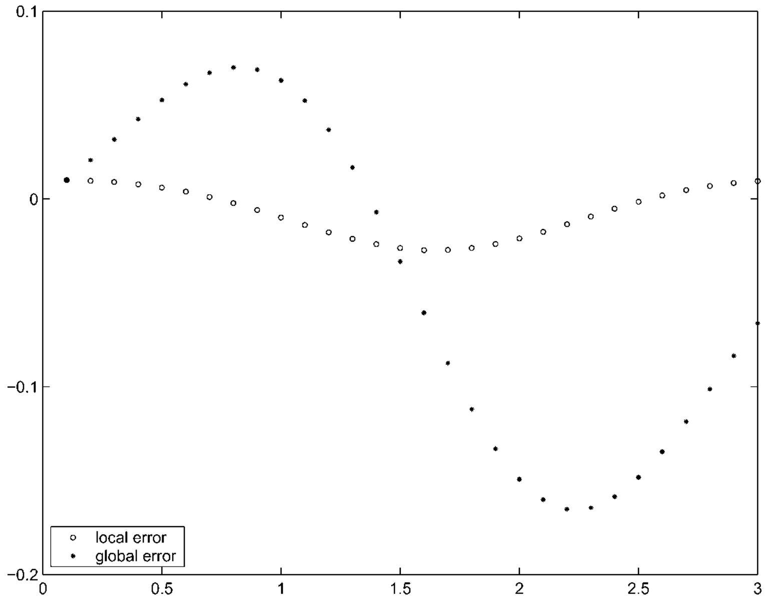

Figure 1.5: Comparison of local and global errors.

the first step. Thereafter, the global errors grow and decay according the stability of the problem, as seen in Figure 1.4.

Backward error analysis has been an invaluable tool for understanding issues arising in numerical linear algebra. It provides a complementary view of numerical methods for ODEs that is especially important for the methods of the Matlab solvers. All these solvers produce approximate solutions $S(t)$ on the whole interval $[a, b]$ that are piecewise smooth. For conceptual purposes, we can define a piecewise-smooth function $S(t)$ with $S(t_n) = y_n$ for each value n that plays the same role for methods that do not naturally produce such an approximation. The *residual* of such an approximation is

$$r(t) = S'(t) - f(t, S(t))$$

Put differently, $S(t)$ is the *exact* solution of the perturbed ODE

$$S'(t) = f(t, S(t)) + r(t)$$

In the view of backward error analysis, $S(t)$ is a good approximate solution if it satisfies an ODE that is “close” to the one given – that is, if the residual $r(t)$ is “small”. This is a perfectly reasonable definition of a “good” solution, but if the IVP is well-posed then it also implies that $S(t)$ is “close” to the true solution $y(t)$, the usual definition of a good approximation. In this view, a solver tries to produce an approximate solution with a small residual. The BVP solver of Matlab does exactly this, and the IVP and DDE solvers do it indirectly.

■ EXERCISE 1.1

Among the examples available through the Matlab command `help dsolve` is the IVP

$$(y')^2 + y^2 = 1, \quad y(0) = 0$$

In addition to showing how easy it is to solve simple IVPs analytically, the example has interesting output:

```
>> y = dsolve('(Dy)^2 + y^2 = 1','y(0) = 0')

y = [  sin(t)]
    [ -sin(t)]
```

According to `dsolve`, this IVP has two solutions. Is this correct? If it is, reconcile this with the existence and uniqueness result for IVPs of Section 1.2.

■ EXERCISE 1.2

Prove that the function $f(t, y)$ in

$$y' = f(t, y) = +\sqrt{|y|}$$

does not satisfy a Lipschitz condition on the rectangle $|t| \le 1$, $|y| \le 1$. Show by example that this ODE has more than one solution that satisfies $y(0) = 0$. Show that $f(t, y)$ does satisfy a Lipschitz condition on the rectangle $|t| \le 1$, $0 < \alpha \le y \le 1$. The general result discussed in the text then says that the ODE has only one solution with its initial value in this rectangle.

■ EXERCISE 1.3

The interval on which the solution of an IVP exists depends on the initial conditions. To see this, find the general solution of the following ODEs and consider how the interval of existence depends on the initial condition:

$$y' = \frac{1}{(t-1)(t-2)}$$
$$y' = -3y^{4/3}\sin(t)$$

■ EXERCISE 1.4

The program `dfs.m` that accompanies this book provides a modest capability for computing a direction field and solutions of a scalar ODE, $y' = f(t, y)$. The first argument of `dfs.m` is a string defining $f(t, y)$. In this the independent variable must be called t and the dependent variable must be called y. The second argument is an array `[wL wR wB wT]` specifying a plot window. Specifically, solutions are plotted for values $y(t)$ with $wL \le t \le wR$, $wB \le y \le wT$. The program first plots a direction field. If you then indicate a point in the plot window by placing the cursor there and clicking, it computes and plots the solution of the ODE through this point. Clicking at a point outside the window terminates the run. For example, Figure 1.4 can be reproduced with the command

```
>> dfs('cos(t)*y',[0 12 -6 6]);
```

and clicking at appropriate points in the window. Use `dfs.m` to study graphically the stability of the ODE (1.11). A plot window appropriate for the IVP studied analytically in the text is given by `[0 5 -2 20]`.

■ EXERCISE 1.5

Compare local and global errors as in Figure 1.5 when solving equation (1.11) with $y(0) = 0.08$. Use Euler's method with the constant step size $h = 0.1$ to integrate from 0 to 2. The stability of this problem is studied analytically in the text and numerically in Exercise 1.4. With this in mind, discuss the behavior of the global errors.

1.3 Standard Form

Ordinary differential equations arise in the most diverse forms. In order to solve an ODE problem, you must first write it in a form acceptable to your code. By far the most common form accepted by IVP solvers is the system of first-order equations discussed in Section 1.2,

$$y' = f(t, y) \tag{1.15}$$

The Matlab IVP solvers accept ODEs of the more general form

$$M(t, y)y' = F(t, y) \tag{1.16}$$

involving a nonsingular *mass matrix* $M(t, y)$. These equations can be written in the form (1.15) with $f(t, y) = M(t, y)^{-1}F(t, y)$, but for some kinds of problems the form (1.16) is more convenient and more efficient. With either form, we must formulate the ODEs as a system of first-order equations. The usual way to do this is to introduce new dependent variables. You must introduce a new variable for each of the dependent variables in the original form of the problem. In addition, a new variable is needed for each derivative of an original variable up to one less than the highest derivative appearing in the original equations. For each new variable, you need an equation for its first derivative expressed in terms of the new variables. A little manipulation using the definitions of the new variables and the original equations is then required to write the new equations in the form (1.15) (or (1.16)). This is harder to explain in words than it is to do, so let's look at some examples. To put the ODE (1.6) describing the motion of a pendulum in standard form, we begin with a new variable $y_1(t) = \theta(t)$. The second derivative of $\theta(t)$ appears in the equation, so we need to introduce one more new variable, $y_2(t) = \theta'(t)$. For these variables we have

$$\begin{aligned} y_1'(t) &= \theta'(t) = y_2(t) \\ y_2'(t) &= \theta''(t) = -\sin(\theta(t)) = -\sin(y_1(t)) \end{aligned}$$

From this we recognize that

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= -\sin(y_1) \end{aligned}$$

that is, the two components of the vector function $f(t, y)$ of (1.15) are given by $f_1(t, y) = y_2$ and $f_2(t, y) = -\sin(y_1)$. When we solved an IVP for this ODE we specified initial values

$$\begin{aligned} y_1(0) &= \theta(0) = 0 \\ y_2(0) &= \theta'(0) \end{aligned}$$

and when we solved a BVP we specified boundary values

$$\begin{aligned} y_1(0) &= \theta(0) = 0 \\ y_1(b) &= \theta(b) = \pi \end{aligned}$$

As another example consider Kepler's equations describing the motion of one body around another of equal mass located at the origin under the influence of gravity. In appropriate units they have the form

$$x'' = -\frac{x}{r^3}, \quad y'' = -\frac{y}{r^3} \tag{1.17}$$

where $r = \sqrt{x^2 + y^2}$. Here $(x(t), y(t))$ are the coordinates of the moving body relative to the body fixed at the origin. With initial values

$$x(0) = 1 - e, \quad y(0) = 0, \quad x'(0) = 0, \quad y'(0) = \sqrt{\frac{1+e}{1-e}} \tag{1.18}$$

there is an analytical solution in terms of solutions of Kepler's (algebraic) equation that shows the orbit is an ellipse of eccentricity e. These equations are easily written as a first-order system. One choice is to introduce variables $y_1 = x$ and $y_2 = y$ for the unknowns and then, because the second derivatives of the unknowns appear in the equations, to introduce variables $y_3 = x'$ and $y_4 = y'$ for their first derivatives. You should verify that the first-order system is

$$\begin{aligned} y_1' &= y_3 \\ y_2' &= y_4 \\ y_3' &= -\frac{y_1}{r^3} \\ y_4' &= -\frac{y_2}{r^3} \end{aligned}$$

where $r = \sqrt{y_1^2 + y_2^2}$, and that the initial conditions are

$$y_1(0) = 1 - e, \quad y_2(0) = 0, \quad y_3(0) = 0, \quad y_4(0) = \sqrt{\frac{1+e}{1-e}}$$

Both of these examples illustrate the fact that mechanical problems described by Newton's laws of motion lead to systems of second-order equations and, if there is no dissipation, there are no first derivatives. Equations like this are called *special second-order equations*. They are sufficiently common that some codes accept IVPs in the standard form

$$y'' = f(t, y)$$

with initial position $y(a)$ and initial velocity $y'(a)$ given. As we have seen, it is easy enough to write such problems as first-order systems, but since there are numerical methods that take advantage of the special form it is both efficient and convenient to work directly with the system of second-order equations (cf. Brankin et al. 1989).

Sometimes it is useful to introduce additional unknowns in order to compute quantities related to the solution. An example arises in formulating the solution of the Sturm–Liouville eigenproblem consisting of the ODE

$$y''(x) + \lambda y(x) = 0$$

with boundary conditions $y(0) = 0$ and $y(2\pi) = 0$. The task is to find an eigenvalue λ for which there is a nontrivial (i.e., not identically zero) solution, known as an eigenfunction. For some purposes it is appropriate to normalize the solution so that

$$1 = \int_0^{2\pi} y^2(t)\, dt$$

A convenient way to impose this normalizing condition is to introduce a variable

$$y_3(x) = \int_0^x y^2(t)\,dt$$

Then, along with variables $y_1(x) = y(x)$ and $y_2(x) = y'(x)$, we have the first-order system

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= -\lambda y_1 \\ y_3' &= y_1^2 \end{aligned}$$

The definition of the new variable implies that $y_3(0) = 0$, and we seek a solution of the system of ODEs for which $y_3(2\pi) = 1$. All together we have three equations and one unknown parameter λ. The solution of interest is to be determined by the four boundary conditions

$$y_1(0) = 0, \quad y_1(2\pi) = 0, \quad y_3(0) = 0, \quad y_3(2\pi) = 1$$

Here we use the device of introducing a new variable for an auxiliary quantity to determine a solution of interest. Another application is to put the problem in standard form. The Matlab BVP solver `bvp4c` accepts problems with unknown parameters, but this facility is not commonly available. Most BVP solvers require that the parameter λ be replaced by a variable $y_4(t)$. The parameter is constant, so the new unknown satisfies the ODE

$$y_4' = 0$$

In this way we obtain a system of four first-order ODEs that does not explicitly involve an unknown parameter,

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= -y_4 y_1 \\ y_3' &= y_1^2 \\ y_4' &= 0 \end{aligned}$$

and the boundary conditions are unchanged. Exercises 1.8 and 1.9 exploit this technique of converting integral constraints to differential equations.

Often a proper selection of unknowns is key to solving a problem. The following example arose in an investigation by chemical engineer F. Song (pers. commun.) into the corrosion of natural gas pipelines under a coating with cathodic protection. The equations are naturally formulated as

$$\frac{d^2x}{dz^2} = \gamma(e^x + \mu_c e^{x\omega_{\mathrm{Fe}}} + \lambda_H e^{x\omega_{\mathrm{H}}} + \lambda_{\mathrm{O}_2} e^{x\omega_{\mathrm{O}_2}})$$
$$\frac{d^2p_{\mathrm{O}_2}}{dz^2} = \pi p_{\mathrm{O}_2} e^{x\omega_{\mathrm{O}_2}} + \beta p_{\mathrm{O}_2} + \kappa$$

This is a BVP with boundary conditions at the origin and infinity. It is possible to eliminate the variable $p_{\mathrm{O}_2}(z)$ to obtain a fourth-order equation for the solution variable $x(z)$ alone. Reducing a set of ODEs to a single, higher-order equation is often useful for analysis, but to solve the problem numerically the equation must then be reformulated as a system of first-order equations. If you forget about the origin of the fourth-order ODE for $x(z)$ here, you might reasonably introduce new variables in the usual way,

$$y_1 = x, \quad y_2 = x', \quad y_3 = x'', \quad y_4 = x'''$$

This is not a good idea because it does not directly account for the behavior of the corrodant, $p_{\mathrm{O}_2}(z)$. It is much better practice here to start with the original formulation and introduce the new variables

$$w_1 = x, \quad w_2 = x', \quad w_3 = p_{\mathrm{O}_2}, \quad w_4 = p'_{\mathrm{O}_2}$$

It is easier to select appropriate error tolerances for quantities that can be interpreted physically. Also, by specifying error tolerances for w_3, we require the solver to compute accurately the fundamental quantity p_{O_2}. When solving BVPs you must provide a guess for the solution. It is easier to provide a reasonable guess for quantities that have physical significance. In Song's work, a suitable formulation of this problem and a corresponding guess was important to the successful solution of this BVP. It is worth noting that here "solving" the problem was not just a matter of computing the solution of a single BVP. As is so often the case in practice, the BVP was to be solved for a range of parameter values.

■ EXERCISE 1.6

Consider the two-point BVP consisting of the second-order ODE

$$(p(x)y')' + q(x)y = r(x)$$

with boundary conditions

$$y(0) = 0, \quad p(1)y'(1) = 2$$

The function $p(x)$ is differentiable and positive for all $x \in [0, 1]$. Using $p'(x)$, write this problem in the form of a first-order system using as unknowns $y_1 = y$ and $y_2 = y'$. In applications it is often natural to use the flux py' as an unknown instead of y'. Indeed, one of the boundary conditions here states that the flux has a given value. Show that with the flux as an unknown, you can write the problem in the form of a first-order system without needing to differentiate $p(x)$.

■ EXERCISE 1.7

Kamke (1971, p. 598) states that the IVP

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

describes space charge current in a cylindrical capacitor.

- Find two equivalent explicit ODEs in special second-order form.
- Formulate the second-order equations as systems of first-order equations.

■ EXERCISE 1.8

Murphy (1965) extends the classical Falkner–Skan similarity solutions for laminar incompressible boundary layer flows to flows over curved surfaces. He derives a BVP consisting of the ODE

$$f'''' + (\Omega + f)f''' + \Omega f f'' - (2\beta - 1)[f'f'' + \Omega(f')^2] = 0$$

to be solved on $0 \le \eta \le b$ with boundary conditions

$$f(0) = f'(0) = 0, \quad f'(b) = e^{-\Omega b}, \quad f''(b) = -\Omega e^{-\Omega b}$$

Here Ω is a curvature parameter, β is a pressure–gradient parameter, and b is large enough that the exponential terms in the boundary conditions describe the correct asymptotic behavior. Physically significant quantities are the displacement thickness

$$\Delta^* = \int_0^b [1 - f'(\eta)e^{\Omega\eta}]\, d\eta$$

and the momentum thickness

$$\theta = \int_0^b f'(\eta)e^{\Omega\eta}[1 - f'(\eta)e^{\Omega\eta}]\, d\eta$$

Formulate the BVP in terms of a system of first-order equations. Add equations and initial values so that the displacement thickness and the momentum thickness can each be computed along with the solution $f(\eta)$.

■ EXERCISE 1.9

Caughy (1970) describes the large-amplitude whirling of an elastic string by a BVP consisting of the ODE

$$\mu'' + \omega^2\left(\frac{1-\alpha^2}{H}\frac{1}{\sqrt{1+\mu^2}} + \alpha^2\right)\mu = 0$$

and boundary conditions

$$\mu'(0) = 0, \quad \mu'(1) = 0$$

Here α is a physical constant with $0 < \alpha < 1$. Because the whirling frequency ω is to be determined as part of solving the BVP, there must be another boundary condition. Caughy specifies the amplitude ε of the solution at the origin:

$$\mu(0) = \varepsilon$$

An unusual aspect of this problem is that an important constant H is defined in terms of the solution $\mu(x)$ throughout the interval of integration:

$$H = \frac{1}{\alpha^2}\left[1 - (1-\alpha^2)\int_0^1 \frac{dx}{\sqrt{1+\mu^2(x)}}\right]$$

Formulate this BVP in standard form. As in the Sturm–Liouville example, you can introduce a new variable $y_3(x)$, a first-order ODE, and a boundary condition to deal with the integral term in the definition of H. The trick to dealing with H is to let it be a new variable $y_4(x)$. It is a constant, so this new variable satisfies the first-order differential equation $y_4' = 0$. It is given the correct constant value by the boundary condition resulting from the definition of H:

$$y_4(1) = \frac{1}{\alpha^2}[1 - (1-\alpha^2)y_3(1)]$$

■ EXERCISE 1.10

This exercise is based on material from the textbook *Continuous and Discrete Signals and Systems* (Soliman & Srinath 1998). A linear, time-invariant (LTI) system is described by a single linear, constant-coefficient ODE of the form

$$y^{(N)}(t) + \sum_{i=0}^{N-1} a_i y^{(i)}(t) = \sum_{i=0}^{N} b_i x^{(i)}(t) \tag{1.19}$$

Here $x(t)$ is a given signal and $y(t)$ is the response of the system. A simulation diagram is a representation of the system using only amplifiers, summers, and integrators. This might be described in many ways, but there are two canonical forms. A state-variable description of a system has some advantages, one being that it is a first-order system of ODEs that is convenient for numerical solution. The two canonical forms for simulation diagrams lead directly to two state-variable descriptions. Let $v(t) = (v_1(t), v_2(t), \ldots, v_N(t))^{\mathrm{T}}$ be a vector of state variables. The description corresponding to the first canonical form is

$$v'(t) = \begin{pmatrix} -a_{N-1} & 1 & 0 & \dots & 0 \\ -a_{N-2} & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ -a_1 & 0 & 0 & \dots & 1 \\ -a_0 & 0 & 0 & \dots & 0 \end{pmatrix} v(t) + \begin{pmatrix} b_{N-1} - a_{N-1}b_N \\ b_{N-2} - a_{N-2}b_N \\ \vdots \\ b_1 - a_1 b_N \\ b_0 - a_0 b_N \end{pmatrix} x(t)$$

The output $y(t)$ is obtained from the equation

$$y(t) = (1, 0, \dots, 0)^{\mathrm{T}} v(t) + b_N x(t)$$

Show directly that you can solve the ODE (1.19) by solving this system of first-order ODEs. Keep in mind that all the coefficients are constant. *Hint:* Using the identity

$$y(t) = v_1(t) + b_N x(t)$$

rewrite the equations so that, for $i < N$,

$$v_i'(t) = (b_{N-i}x(t) - a_{N-i}y(t)) + v_{i+1}(t)$$

Differentiate the equation for $v_1'(t)$ and use the equation for $v_2'(t)$ to obtain an equation for $v_1''(t)$ involving $v_3(t)$. Repeat until you have an equation for $v_1^{(N)}(t)$, equate it to $(y(t) - b_N x(t))^{(N)}$, and compare the result to the ODE (1.19).

The description corresponding to the second canonical form is

$$v'(t) = \begin{pmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1 \\ -a_0 & -a_1 & -a_2 & \dots & -a_{N-1} \end{pmatrix} v(t) + \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix} x(t)$$

Obtaining the output is more complicated for this form. The formula is

$$y(t) = [(b_0 - a_0 b_N), (b_1 - a_1 b_N), \dots, (b_{N-1} - a_{N-1}b_N)]^{\mathrm{T}} v(t) + b_N x(t)$$

Show directly that you can solve the ODE (1.19) by solving this system of first-order ODEs. *Hint:* Define the function $w(t)$ as the solution of the ODE

$$w^{(N)}(t) + \sum_{j=0}^{N-1} a_j w^{(j)}(t) = x(t)$$

and then show by substitution that the function

$$y(t) = \sum_{i=0}^{N} b_i w^{(i)}(t)$$

satisfies the ODE (1.19). Finally, obtain a set of first-order ODEs for the function $w(t)$ in the usual way.

It is striking that the derivatives $x^{(i)}(t)$ do not appear in either of the two canonical systems. Show that they play a role when you want to find a set of initial conditions $v_i(0)$ that corresponds to a set of initial conditions for $y^{(i)}(0)$ and $x^{(i)}(0)$ in the original variables.

1.4 Control of the Error

ODE solvers ask how much accuracy you want because the more you want, the more the computation will cost. The Matlab solvers have error tolerances in the form of a scalar relative error tolerance re and a vector of absolute error tolerances ae. The solvers produce vectors $y_n = (y_{n,i})$ that approximate the solution $y(t_n) = (y_i(t_n))$ on the mesh (1.9). Stated superficially, at each point in the mesh they aim to produce an approximation that satisfies

$$|y_i(t_n) - y_{n,i}| \le re|y_i(t_n)| + ae_i \tag{1.20}$$

for each component of the solution. Variants of this kind of control are seen in all the popular IVP solvers. For the convenience of users, the Matlab solvers interpret a scalar absolute error tolerance as applying to all components of the solution. Also for convenience, default error tolerances are supplied. They are 10^{-3} for the relative error tolerance and a scalar 10^{-6} for the absolute error tolerance. The default relative error tolerance has this value because solutions are usually interpreted graphically in Matlab. A relative error tolerance of 10^{-5} is more typical of general scientific computing.

For a code with a vector of relative error tolerances RTOL and a vector of absolute error tolerances ATOL, Brenan, Campbell, & Petzold (1996, p. 131) state:

> *We cannot emphasize strongly enough the importance of carefully selecting these tolerances to accurately reflect the scale of the problem.* In particular, for problems whose solution components are scaled very differently from each other, it is advisable to provide the code with vector valued tolerances. For users who are not sure how to set the tolerances RTOL and ATOL, we recommend starting with the following rule of thumb. Let m be the number of significant digits required for solution component y_i. Set $\text{RTOL}_i = 10^{-(m+1)}$. Set ATOL_i to the value at which $|y_i|$ is essentially insignificant.

Because we agree about the importance of selecting appropriate error tolerances, we have devoted this section to a discussion of the issues. This discussion will help you understand the rule of thumb.

The inequality (1.20) defines a *mixed* error control. If all the values $ae_i = 0$, it corresponds to a pure *relative* error control; if the value $re = 0$, it corresponds to a pure *absolute* error control. The pure error controls expose more clearly the roles of the two kinds of tolerances and the difficulties associated with them. First suppose that we use a pure relative error control. It requires that

$$\left| \frac{y_i(t_n) - y_{n,i}}{y_i(t_n)} \right| \le re$$

for each solution component. There are two serious difficulties. One is that a pure relative error control is not appropriate if the solution might vanish. The formal difficulty is that the denominator $y_i(t_n)$ might vanish. However, we are attempting to control the error in a function, so the more fundamental question is: What should we mean by relative error if $y_i(t)$ might vanish at some isolated point $t = t^*$? The solvers commonly compare the error to some measure of the size of $y_i(t)$ near t_n rather than just the value $|y_i(t_n)|$ of (1.20). This is a reasonable and effective approach, but it does not deal with a component $y_i(t)$ that is zero throughout an interval about t_n. Solvers must therefore recognize the possibility that a relative error control is not well-defined, even in some extended sense, and terminate the integration with a message should this occur. You can avoid the difficulty by specifying a nonzero absolute error tolerance in a mixed error test. For robustness some solvers, including those of Matlab, require that absolute error tolerances be positive.

Before taking up the other difficulty, we need to make some comments about computer arithmetic. Programming languages like Fortran 77 and C include both single and double precision arithmetic. Typically this corresponds to about 7 and 16 decimal digits, respectively. Matlab has only one precision, typically double precision. Experience says that, when solving IVPs numerically, it is generally best to use double precision. The floating point representation of a number is accurate only to a unit roundoff, which is determined by the working precision. In Matlab it is called `eps` and for a PC it is typically $2.2204 \cdot 10^{-16}$, corresponding to double precision in the IEEE-754 definition of computer arithmetic that is used almost universally on today's computers. Throughout this book we assume that the unit roundoff is about this size when we speak of computations in Matlab.

A relative error tolerance specifies roughly how many correct digits you want in an answer. It makes no sense to ask for an answer more accurate than the floating point representation of the true solution – that is, it is not meaningful to specify a value re smaller than a unit roundoff. Of course, a tolerance that is close to a unit roundoff is usually also too small because finite precision arithmetic affects the computation and hence the accuracy that a numerical method can deliver. For this reason the Matlab solvers require that re be larger than a smallish multiple of `eps`, with the multiple depending on the particular solver. You might expect that a code would fail in some dramatic way if you ask for an impossible accuracy. Unfortunately, that is generally not the case. If you experiment with a code that does not check then you are likely to find that, as you decrease the tolerances past the point where you are requesting an impossible accuracy: the cost of the integration increases rapidly; the results are increasingly *less* accurate; and there is no indication from the solver that it is having trouble, other than the increase in cost.

Now we turn to a pure absolute error control. It requires that

$$|y_i(t_n) - y_{n,i}| \le ae_i$$

for each solution component. The main difficulty with an absolute error control is that you must make a judgment about the likely sizes of solution components, and you can get into trouble if you are badly wrong. One possibility is that a solution component is much larger in magnitude than expected. A little manipulation of the absolute error control inequality leads to

$$\left| \frac{y_i(t_n) - y_{n,i}}{y_i(t_n)} \right| \le \frac{ae_i}{|y_i(t_n)|}$$

This makes clear that a pure absolute error tolerance of ae_i on $y_i(t)$ corresponds to a relative error tolerance of $ae_i/|y_i(t_n)|$ on this component. If $|y_i(t_n)|$ is sufficiently large, then specifying an absolute error tolerance that seems unremarkable can correspond to asking for an answer that is more accurate in a relative sense than a unit roundoff. As we have just seen, that is an impossible accuracy request. The situation can be avoided by specifying a nonzero relative error tolerance and thus a mixed error control. Again for the sake of robustness, the Matlab solvers do this by requiring that the relative error tolerance be greater than a few units of roundoff.

The other situation that concerns us with pure absolute error control is when a solution component is much smaller than its absolute error tolerance. First we must understand what the error control means for such a component. If (say) $|y_i(t_n)| < 0.5ae_i$, then *any* approximation $y_{n,i}$ for which $|y_{n,i}| < 0.5ae_i$ will pass the error test. Accordingly, an acceptable approximation may have *no* correct digits. You might think that you always need some accuracy, but for many mathematical models of physical processes there are quantities that have negligible effects when they fall below certain thresholds and are then no longer interesting. The danger is that one of these quantities might later grow to the point that it must again be taken into account. If a solution component is rather smaller in magnitude than its absolute error tolerance and if you require some accuracy in this component, you will need to adjust the tolerance and solve the problem again. It is an interesting and useful fact that you may very well compute some correct digits in a "small" component even though you did not require it by means of its error tolerance. One reason is that the solver may have computed this component with some accuracy in order to achieve the accuracy specified for a component that depends on it. Another reason is that the solver selects a step size small enough to deal with the solution component that is most difficult to approximate to within the accuracy specified. Generally this step size is smaller than necessary for other components, so they are computed more accurately than required.

The first example of Lapidus, Aiken, & Liu (1973) is illustrative. Proton transfer in a hydrogen–hydrogen bond is described by the system of ODEs

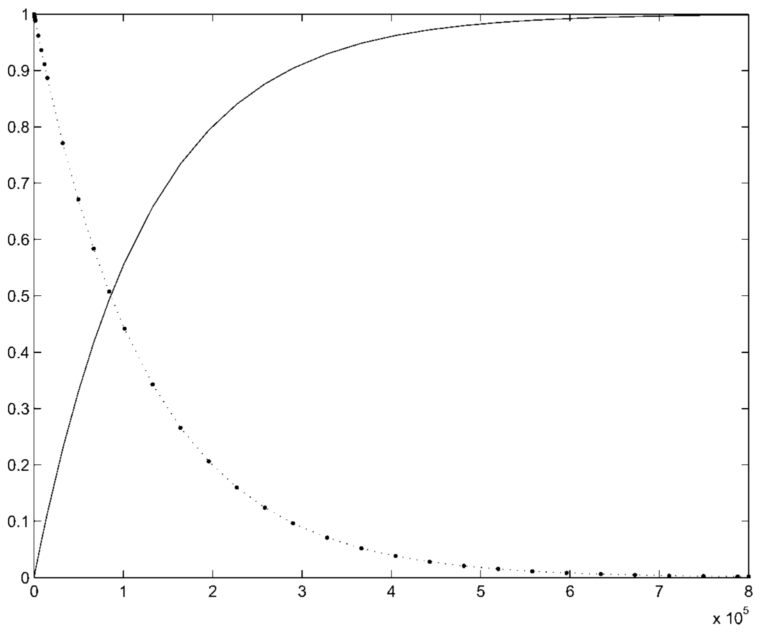

Figure 1.6: Solution components $x_1(t)$ and $x_2(t)$ of the proton transfer problem.

$$
\begin{aligned}
x_1' &= -k_1 x_1 + k_2 y \\
x_2' &= -k_4 x_2 + k_3 y \\
y' &= k_1 x_1 + k_4 x_2 - (k_1 + k_3) y
\end{aligned}
\tag{1.21}
$$

to be solved with initial values

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

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

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

This is an example of a *stiff* problem. We solved it easily with the Matlab IVP solver `ode15s` using default error tolerances, but we found that the quickly reacting intermediate component $y(t)$ is very much smaller than the default absolute error tolerance of 10^{-6}. Despite this, it was computed accurately enough to give a general idea of its size. Once we recognized how small it is, we reduced the absolute error tolerance to 10^{-20} and obtained the solutions displayed in Figures 1.6 and 1.7. It is easy and natural in exploratory computations with the Matlab ODE solvers to display all the solution components on one plot. If some components are invisible then you might want to determine

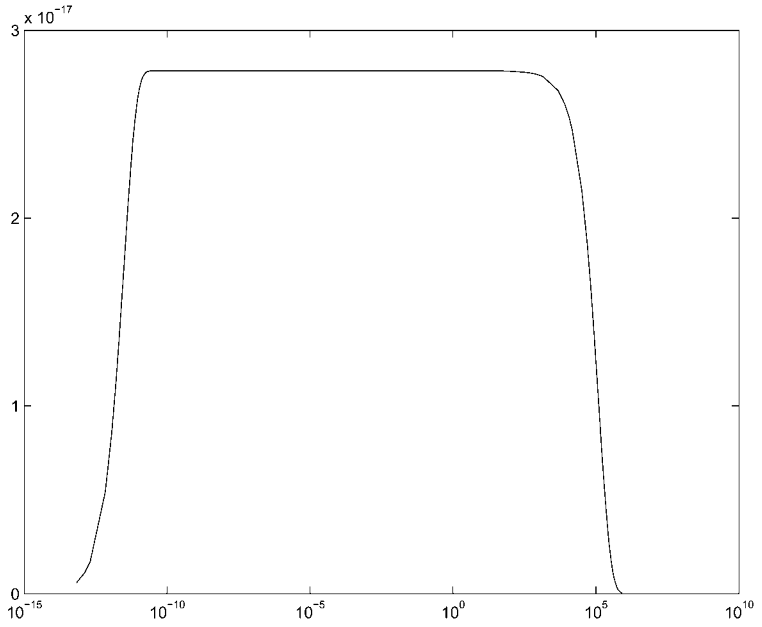

Figure 1.7: Solution $y(t)$ of proton transfer problem, `semilogx` plot.

the maximum magnitudes of the solution components – both to identify components for plotting separately on different scales and for choosing tolerances for another, more accurate computation.

Often in modeling chemical reactions, concentrations that have dropped below a certain threshold have negligible effects and so are of no physical interest. Then it is natural to specify absolute error tolerances of about the sizes of these thresholds. The concentrations $y_i(t)$ are positive, but when tracking a component $y_i(t)$ that decays to zero a solver might generate a "small" solution component $y_{n,i} < 0$. As we have seen, the error control permits this and it sometimes happens. A small negative approximation to a concentration may just be an annoyance, but some models are not stable in these circumstances and the computation blows up. It is ironic that a quantity so small that it is unimportant physically can destroy the numerical solution. An IVP popularized by Robertson (1966) as a test problem for solvers intended for stiff IVPs provides a concrete example. A chemical reaction is described by the system of ODEs

$$\begin{aligned} y_1' &= -0.04y_1 + 10^4 y_2 y_3 \\ y_2' &= 0.04y_1 - 10^4 y_2 y_3 - 3\cdot 10^7 y_2^2 \\ y_3' &= 3\cdot 10^7 y_2^2 \end{aligned} \tag{1.22}$$

with initial conditions

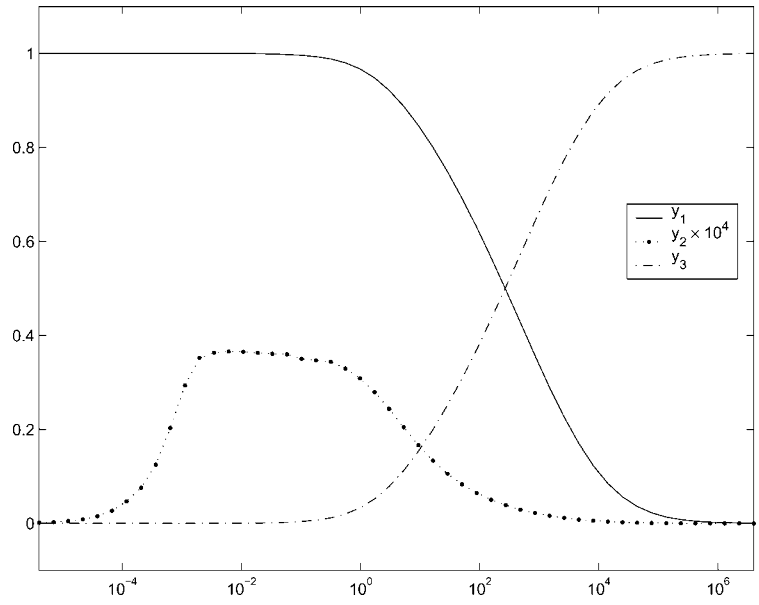

Figure 1.8: Robertson's problem; a `semilogx` plot of the solution.

$$\begin{pmatrix} y_1(0) \\ y_2(0) \\ y_3(0) \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

It is not difficult to show that, for all times $t > 0$, the solution components are nonnegative and sum to 1. This is an example of a linear conservation law that we will discuss in some detail in the next section.

The `hb1ode` demonstration program of Matlab integrates this problem with `ode15s` from time $t = 0$ to near steady state at $t = 4 \cdot 10^6$. A small modification of its output resulted in Figure 1.8. Hindmarsh & Byrne (1976) use this problem to illustrate the performance of their code `EPISODE` for solving stiff IVPs. With a moderately stringent absolute error tolerance of 10^{-6}, they find that if they continue the integration then a small nonphysical negative concentration is computed that begins growing rapidly in magnitude. Soon the numerical solution is completely unacceptable. A portion of one of their tables of results is given in Table 1.1. We emphasize that the unsatisfactory performance is a consequence of the problem and what is asked of the solver; something similar happens when other solvers are used, including `ode15s`. For more details about this example see Hindmarsh & Byrne (1976) and Shampine (1994). Different but related problems are considered in Exercises 1.12 and 1.13.

We have seen that you cannot ask for too much accuracy in a relative sense. We take this opportunity to advise you not to ask for too little. This is a temptation because the

Table 1.1: *Robertson's problem; steady-state solution computed using* `EPISODE`.

t	y_1	y_2	y_3
4e5	4.9394e−03	1.9854e−08	9.9506e−01
4e7	3.2146e−05	1.2859e−10	9.9997e−01
4e9	−1.8616e+06	−4.0000e−06	1.8616e+06

more accuracy you want, the more the computation will cost. It is especially tempting when the data of a problem is known only to a digit or two. (We have solved IVPs for which even the order of magnitude of measured data was in doubt.) Nevertheless, asking for too little accuracy is both dangerous and pointless. The basic algorithms are valid only when the step sizes are sufficiently small. If you do not ask for enough accuracy, a solver might choose step sizes that are too large for reliable results. A quality solver may recognize that it must use smaller step sizes for reliability and in effect reduce the error tolerances that you specify. As explained in Section 1.2, the solvers control local errors and only indirectly control the error in the solution $y(t)$. They maintain these local errors somewhat smaller than the tolerances. How much smaller is "tuned" for the solver so that, for typical IVPs, the error in $y(t)$ is smaller than (or comparable to) the tolerances specified. If your IVP is somewhat unstable or you expect the solution to oscillate often in the interval of interest, then you should be cautious about asking for too little accuracy because you might well be disappointed in the accuracy that you get. That is the least of your worries: You might compute a solution that is not physically realistic, or one that is physically realistic but incorrect, or the computation might fail entirely. In considering this it is important to appreciate that the solver is doing exactly what you tell it to do – namely, to control the local error so that it is no larger than the specified tolerances. Unsatisfactory results are usually a consequence of the instability of the IVP, not of the solver. Figure 1.9 of Section 1.5 shows what can happen. The dotted curve is an orbit of one body about another that was computed with default error tolerances. These default error tolerances were intended to be satisfactory for plotting the solutions of typical problems, but in this instance the orbit is not even qualitatively correct. Displayed as a solid curve is the same orbit computed with more stringent error tolerances. It is qualitatively correct. Clearly it is important not to ask for too little accuracy when solving this problem.

When solving a newly formulated IVP, it may be necessary to experiment with the choice of error tolerances. To do this, you may need to inspect solutions to verify that you are using an appropriate error control. You may also want to try reducing the error tolerances to verify by consistency that you are asking for sufficient accuracy to reflect the qualitative behavior of the solution.

■ **EXERCISE 1.11**

To simplify their user interface, some codes ask for a single error tolerance τ. For example, `DVERK` (Hull, Enright, & Jackson 1975) requires that, at each step,

$$|y_i(t_n) - y_{n,i}| \le \tau \max(1, |y_i(t_n)|)$$

and `MIRKDC` (Enright & Muir 1996) requires the equivalent of

$$|y_i(t_n) - y_{n,i}| \le \tau(1 + |y_i(t_n)|)$$

Argue that these are roughly equivalent to the error control (1.20) with $re = \tau$ and $ae_i = \tau$ for each i. People sometimes get into trouble with this kind of error control because they do not realize that they are specifying an absolute error tolerance that is not appropriate for the problem they are solving.

■ **EXERCISE 1.12**

The solution of

$$y' = f(t, y) = \sqrt{1 - y^2}, \quad y(0) = 0$$

is $\sin(t)$. When computing this solution numerically, why should you expect to get into trouble as you approach $t = 0.5\pi$? There are two kinds of difficulties, one involving the error control and one involving uniqueness.

■ **EXERCISE 1.13**

If you solve the IVP

$$y' = \left(\frac{2\ln(y) + 8}{t} - 5\right)y, \quad y(1) = 1$$

with a code written in Matlab, you might compute approximations to $y(t)$ that are complex-valued for “large” t. Codes in other computing environments might fail outright. What is going on? To answer this question it is helpful to know that the solution is

$$y(t) = e^{-t^2+5t-4}$$

1.5 Qualitative Properties

We have seen several examples of solutions with certain qualitative properties that are implied by the ODEs. It is commonly assumed that numerical solutions inherit these properties, but with one major exception they do *not*. The best we can say for standard methods is that the numerical solutions have approximately the same behavior as the analytical solutions. There are ways of making standard methods do better in this regard and there are methods that preserve certain qualitative properties, but we do not pursue such specialized aspects of solving ODEs in this book. For further information about these matters you might turn to Sanz-Serna & Calvo (1994), Shampine (1986), Stuart & Humphries (1996), and the references therein.

We begin our discussion of qualitative properties with one that *is* inherited by virtually all standard methods. If there is a constant (column) vector c such that $c^{\mathrm{T}} f(t, y) \equiv 0$, then the solution of the ODE system

$$y' = f(t, y), \quad y(a) = A$$

satisfies the *linear conservation law,*

$$c^{\mathrm{T}} y(t) \equiv c^{\mathrm{T}} A$$

This follows on observing that

$$\frac{d}{dt}(c^{\mathrm{T}} y(t)) = c^{\mathrm{T}} y'(t) = c^{\mathrm{T}} f(t, y(t)) \equiv 0$$

and hence $c^{\mathrm{T}} y(t)$ is constant. Linear conservation laws express physical laws such as conservation of mass and charge balance. The hydrogen–hydrogen bond problem (1.21) and Robertson's problem (1.22) are examples. With the initial values specified, the solutions of both these problems have components that sum to 1. As it turns out (Shampine 1998), all the standard numerical methods for IVPs preserve all linear conservation laws. For example, if the components of the solution sum to 1, then so do the components of the numerical approximation (to within roundoff errors). The fact that the numerical solution satisfies one or more conservation laws does not mean that it is accurate – even the terrible numerical solution of Robertson's problem found in Table 1.1 has components that sum to 1. On the other hand, if a linear conservation law is not satisfied by the numerical solution to roundoff level, then there is a bug in the program that produced it or the computations were overwhelmed by the effects of finite precision arithmetic. We turn now to properties that are not preserved by standard methods.

In Section 1.2 we found that solutions of the pendulum equation (1.6) have a constant energy. Generally the numerical solutions computed with standard software have an energy that is only approximately constant. To see that it is at least approximately constant, suppose that the equation is written as a first-order system. Further suppose that, at time t_n, the solver produces approximations

$$y_{n,1} = \theta(t_n) + e_1, \quad y_{n,2} = \theta'(t_n) + e_2$$

with small errors e_1 and e_2. By linearization we approximate the energy of the numerical solution as

$$
\begin{aligned}
0.5(y_{n,2})^2 - \cos(y_{n,1}) &= 0.5(\theta'(t_n) + e_2)^2 - \cos(\theta(t_n) + e_1) \\
&\approx E + \theta'(t_n)e_2 + \sin(\theta(t_n))e_1
\end{aligned}
$$

This tells us that the error in the energy is comparable to the errors in the solution components; hence the energy is approximately constant. Often this is satisfactory. However, the long-term qualitative behavior of solutions may depend on the energy and it may be important to conserve energy. One way to do this is simply to solve the equations very accurately using a standard code. This may be satisfactory for short to medium time scales. Alternatively, there are codes based on standard methods that optionally perturb the numerical solution so that it satisfies specified nonlinear conservation laws. There are also codes based on methods that automatically conserve certain physically important quantities, usually energy and/or angular momentum. Whether it is more efficient to use one of these specialized codes or to ask for more accuracy from a standard code is a matter for experimentation. In many cases conservation of a nonlinear conservation law may only be achieved at a high cost or at the expense of accuracy in the solution.

Solutions of the two-body problem (1.17) satisfy two nonlinear conservation laws. The energy

$$\frac{x'(t)^2 + y'(t)^2}{2} - \frac{1}{r(t)}$$

(where the distance $r(t) = \sqrt{x(t)^2 + y(t)^2}$) and the angular momentum

$$x(t)y'(t) - y(t)x'(t)$$

are constant. Figure 1.9 shows the solution of the ODE system (1.17) with initial conditions (1.18) when the eccentricity $e = 0.9$. The path of the moving body displayed as a solid curve was computed with moderately stringent tolerances. The other path was computed with default error tolerances. The fixed body at the origin is shown as an asterisk. For this problem and particular choice of integrator, the energy of the numerical solution computed with default error tolerances decreases steadily from -0.5000 to -0.7874 while the angular momentum decreases from 0.4359 to 0.3992. A steady loss of energy in the physical problem corresponds to the moving body spiraling in to the fixed body. What happens in a numerical computation will depend on the method used and the details of the problem. The point, however, is that the numerical solution satisfies the conservation laws only approximately. Over a time interval sufficiently long, the numerical solution might have a behavior that is qualitatively different from the mathematical solution. Because this particular integrator is losing energy steadily for this particular problem, the effect is pronounced. On the other hand, when we tell the integrator to compute a more accurate answer by specifying smaller error tolerances, we compute a solution on $[0, 20]$ that has the expected behavior. There has been a small loss of energy by time $t = 20$ in this integration, but it is too small for the effect on the computed solution to be visible in the plot.

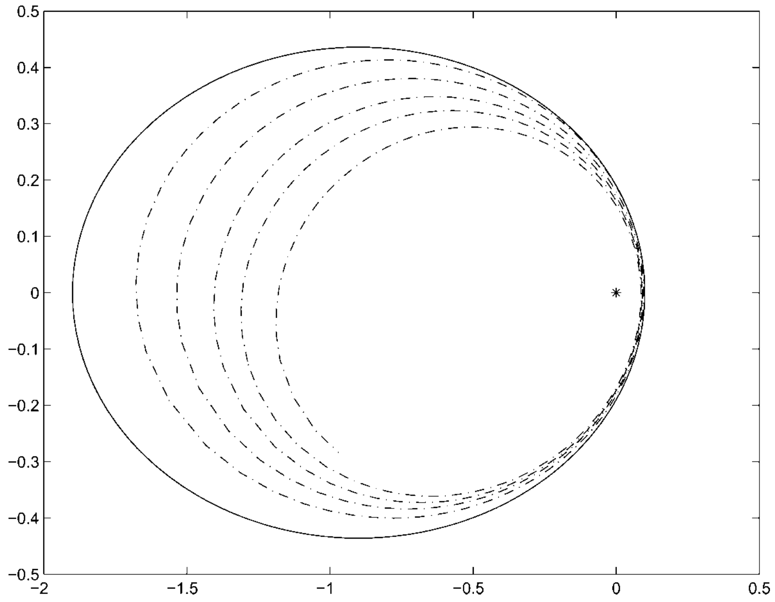

Figure 1.9: Two-body problem for $e = 0.9$ and $0 \le t \le 20$.

There are methods that preserve (at least approximately) certain qualitative properties of IVPs over extended integrations. For example, for ODEs that define symplectic or time-reversible maps, it is possible to construct numerical methods with the corresponding property. These methods bound the error in the Hamiltonian energy and, in some cases, conserve angular momentum; see Sanz-Serna & Calvo (1994) or Stuart & Humphries (1996) for details. Of course, these desirable properties come at a price. The additional constraints placed on the methods to achieve a special property such as symplecticness potentially reduce the accuracy that can be achieved in the computation of the solution at a given cost.

■ EXERCISE 1.14

The differential equations

$$\begin{aligned} y_1' &= -y_1 \\ y_k' &= (k-1)y_{k-1} - k y_k \quad \text{for } k = 2, 3, \ldots, 9 \\ y_{10}' &= 9y_9 \end{aligned}$$

describe the evolution of a chemical reaction. Show that this system of ODEs satisfies a linear conservation law. Specifically, show that the sum of the solution components is constant.

■ EXERCISE 1.15

Volterra's model of predator–prey interaction can be formulated as

$$x' = a(x - xy)$$
$$y' = -c(x - xy)$$

- Show that solutions of this system of ODEs satisfy the nonlinear conservation law

$$G(t, x, y) = x^{-c} y^{-a} e^{cx+ay} = \text{constant}$$

- Write a Matlab program to integrate the differential equations with Euler's method and constant step size h. Using parameter values $a = 2$ and $c = 1$ and initial values $x(0) = 1$ and $y(0) = 3$, integrate the IVP for $0 \leq t \leq 10$. Plot the solution in the phase plane; that is, plot $(x(t), y(t))$. Also, calculate and plot the conserved quantity $G(t, x(t), y(t))$. The theory says that G is constant and the solution is periodic, hence the curve plotted in the phase plane is closed. Experiment with the step size h to find a value for which G is approximately constant and the curve you compute appears to be closed. After you have learned to use the Matlab IVP solvers in the next chapter, you may want to revisit this problem and solve it with `ode45` instead of Euler's method.