Chapter 7

Ordinary Differential Equations

MATLAB has several different functions for the numerical solution of ordinary differential equations. This chapter describes the simplest of these functions and then compares all of the functions for efficiency, accuracy, and special features. Stiffness is a subtle concept that plays an important role in these comparisons.

7.1 Integrating Differential Equations

The *initial value problem* for an ordinary differential equation involves finding a function $y(t)$ that satisfies

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

together with the initial condition

$$y(t_0) = y_0.$$

A numerical solution to this problem generates a sequence of values for the independent variable, $t_0, t_1, \ldots$, and a corresponding sequence of values for the dependent variable, $y_0, y_1, \ldots$, so that each y_n approximates the solution at t_n:

$$y_n \approx y(t_n),\ n = 0, 1, \ldots .$$

Modern numerical methods automatically determine the step sizes

$$h_n = t_{n+1} - t_n$$

so that the estimated error in the numerical solution is controlled by a specified tolerance.

The fundamental theorem of calculus gives us an important connection between differential equations and integrals:

$$y(t + h) = y(t) + \int_t^{t+h} f(s, y(s))ds.$$

December 26, 2005

We cannot use numerical quadrature directly to approximate the integral because we do not know the function $y(s)$ and so cannot evaluate the integrand. Nevertheless, the basic idea is to choose a sequence of values of h so that this formula allows us to generate our numerical solution.

One special case to keep in mind is the situation where $f(t, y)$ is a function of t alone. The numerical solution of such simple differential equations is then just a sequence of quadratures:

$$y_{n+1} = y_n + \int_{t_n}^{t_{n+1}} f(s)ds.$$

Throughout this chapter, we frequently use "dot" notation for derivatives:

$$\dot{y} = \frac{dy(t)}{dt} \text{ and } \ddot{y} = \frac{d^2y(t)}{dt^2}.$$

7.2 Systems of Equations

Many mathematical models involve more than one unknown function, and second- and higher order derivatives. These models can be handled by making $y(t)$ a vector-valued function of t. Each component is either one of the unknown functions or one of its derivatives. The MATLAB vector notation is particularly convenient here.

For example, the second-order differential equation describing a simple harmonic oscillator

$$\ddot{x}(t) = -x(t)$$

becomes two first-order equations. The vector $y(t)$ has two components, $x(t)$ and its first derivative $\dot{x}(t)$:

$$y(t) = \begin{bmatrix} x(t) \\ \dot{x}(t) \end{bmatrix}.$$

Using this vector, the differential equation is

$$\begin{aligned} \dot{y}(t) &= \begin{bmatrix} \dot{x}(t) \\ -x(t) \end{bmatrix} \\ &= \begin{bmatrix} y_2(t) \\ -y_1(t) \end{bmatrix}. \end{aligned}$$

The MATLAB function defining the differential equation has t and y as input arguments and should return $f(t, y)$ as a column vector. For the harmonic oscillator, the function is an M-file containing

```
function ydot = harmonic(t,y)
ydot = [y(2); -y(1)]
```

A fancier version uses matrix multiplication in an inline function,

```
f = inline('[0 1; -1 0]*y','t','y');
```

or an anonymous function,

```
f = @(t,y) [0 1; -1 0]*y
```

In all cases, the variable `t` has to be included as the first argument, even though it is not explicitly involved in the differential equation.

A slightly more complicated example, the *two-body problem*, describes the orbit of one body under the gravitational attraction of a much heavier body. Using Cartesian coordinates, $u(t)$ and $v(t)$, centered in the heavy body, the equations are

$$\ddot{u}(t) = -u(t)/r(t)^3,$$
$$\ddot{v}(t) = -v(t)/r(t)^3,$$

where

$$r(t) = \sqrt{u(t)^2 + v(t)^2}.$$

The vector $y(t)$ has four components:

$$y(t) = \begin{bmatrix} u(t) \\ v(t) \\ \dot{u}(t) \\ \dot{v}(t) \end{bmatrix}.$$

The differential equation is

$$\dot{y}(t) = \begin{bmatrix} \dot{u}(t) \\ \dot{v}(t) \\ -u(t)/r(t)^3 \\ -v(t)/r(t)^3 \end{bmatrix}.$$

The MATLAB function could be

```
function ydot = twobody(t,y)
r = sqrt(y(1)^2 + y(2)^2);
ydot = [y(3); y(4); -y(1)/r^3; -y(2)/r^3];
```

A more compact MATLAB function is

```
function ydot = twobody(t,y)
ydot = [y(3:4); -y(1:2)/norm(y(1:2))^3]
```

Despite the use of vector operations, the second M-file is not significantly more efficient than the first.

7.3 Linearized Differential Equations

The local behavior of the solution to a differential equation near any point (t_c, y_c) can be analyzed by expanding $f(t, y)$ in a two-dimensional Taylor series:

$$f(t, y) = f(t_c, y_c) + \alpha(t - t_c) + J(y - y_c) + \cdots,$$

where

$$\alpha = \frac{\partial f}{\partial t}(t_c, y_c), \quad J = \frac{\partial f}{\partial y}(t_c, y_c).$$

The most important term in this series is usually the one involving J, the Jacobian. For a system of differential equations with n components,

$$\frac{d}{dt}\begin{bmatrix} y_1(t) \\ y_2(t) \\ \vdots \\ y_n(t) \end{bmatrix} = \begin{bmatrix} f_1(t, y_1, \ldots, y_n) \\ f_2(t, y_1, \ldots, y_n) \\ \vdots \\ f_n(t, y_1, \ldots, y_n) \end{bmatrix},$$

the Jacobian is an n-by-n matrix of partial derivatives:

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial y_1} & \frac{\partial f_1}{\partial y_2} & \ldots & \frac{\partial f_1}{\partial y_n} \\ \frac{\partial f_2}{\partial y_1} & \frac{\partial f_2}{\partial y_2} & \ldots & \frac{\partial f_2}{\partial y_n} \\ \vdots & \vdots & & \vdots \\ \frac{\partial f_n}{\partial y_1} & \frac{\partial f_n}{\partial y_2} & \ldots & \frac{\partial f_n}{\partial y_n} \end{bmatrix}.$$

The influence of the Jacobian on the local behavior is determined by the solution to the linear system of ordinary differential equations

$$\dot{y} = Jy.$$

Let $\lambda_k = \mu_k + i\nu_k$ be the eigenvalues of J and $\Lambda = \text{diag}(\lambda_k)$ the diagonal eigenvalue matrix. If there is a linearly independent set of corresponding eigenvectors V, then

$$J = V\Lambda V^{-1}.$$

The linear transformation

$$Vx = y$$

transforms the local system of equations into a set of decoupled equations for the individual components of x:

$$\dot{x}_k = \lambda_k x_k.$$

The solutions are

$$x_k(t) = e^{\lambda_k(t-t_c)} x(t_c).$$

A single component $x_k(t)$ grows with t if μ_k is positive, decays if μ_k is negative, and oscillates if ν_k is nonzero. The components of the local solution $y(t)$ are linear combinations of these behaviors.

For example, the harmonic oscillator

$$\dot{y} = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} y$$

is a linear system. The Jacobian is simply the matrix

$$J = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}.$$

The eigenvalues of J are $\pm i$ and the solutions are purely oscillatory linear combinations of e^{it} and e^{-it}.

A nonlinear example is the two-body problem

$$\dot{y}(t) = \begin{bmatrix} y_3(t) \\ y_4(t) \\ -y_1(t)/r(t)^3 \\ -y_2(t)/r(t)^3 \end{bmatrix},$$

where

$$r(t) = \sqrt{y_1(t)^2 + y_2(t)^2}.$$

In exercise 7.6, we ask you to show that the Jacobian for this system is

$$J = \frac{1}{r^5}\begin{bmatrix} 0 & 0 & r^5 & 0 \\ 0 & 0 & 0 & r^5 \\ 2y_1^2 - y_2^2 & 3y_1y_2 & 0 & 0 \\ 3y_1y_2 & 2y_2^2 - y_1^2 & 0 & 0 \end{bmatrix}.$$

It turns out that the eigenvalues of J just depend on the radius $r(t)$:

$$\lambda = \frac{1}{r^{3/2}}\begin{bmatrix} \sqrt{2} \\ i \\ -\sqrt{2} \\ -i \end{bmatrix}.$$

We see that one eigenvalue is real and positive, so the corresponding component of the solution is growing. One eigenvalue is real and negative, corresponding to a decaying component. Two eigenvalues are purely imaginary, corresponding to oscillatory components. However, the overall global behavior of this nonlinear system is quite complicated and is not described by this local linearized analysis.

7.4 Single-Step Methods

The simplest numerical method for the solution of initial value problems is *Euler's* method. It uses a fixed step size h and generates the approximate solution by

$$y_{n+1} = y_n + hf(t_n, y_n),$$
$$t_{n+1} = t_n + h.$$

The MATLAB code would use an initial point `t0`, a final point `tfinal`, an initial value `y0`, a step size `h`, and an inline function or function handle `f`. The primary loop would simply be

```
t = t0;
y = y0;
while t <= tfinal
   y = y + h*feval(f,t,y)
   t = t + h
end
```

Note that this works perfectly well if `y0` is a vector and `f` returns a vector.

As a quadrature rule for integrating $f(t)$, Euler's method corresponds to a rectangle rule where the integrand is evaluated only once, at the left-hand endpoint of the interval. It is exact if $f(t)$ is constant, but not if $f(t)$ is linear. So the error is proportional to h. Tiny steps are needed to get even a few digits of accuracy. But, from our point of view, the biggest defect of Euler's method is that it does not provide an error estimate. There is no automatic way to determine what step size is needed to achieve a specified accuracy.

If Euler's method is followed by a second function evaluation, we begin to get a viable algorithm. There are two natural possibilities, corresponding to the midpoint rule and the trapezoid rule for quadrature. The midpoint analogue uses Euler to step halfway across the interval, evaluates the function at this intermediate point, then uses that slope to take the actual step:

$$\begin{aligned} s_1 &= f(t_n, y_n), \\ s_2 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2} s_1\right), \\ y_{n+1} &= y_n + h s_2, \\ t_{n+1} &= t_n + h. \end{aligned}$$

The trapezoid analogue uses Euler to take a tentative step across the interval, evaluates the function at this exploratory point, then averages the two slopes to take the actual step:

$$\begin{aligned} s_1 &= f(t_n, y_n), \\ s_2 &= f(t_n + h, y_n + h s_1), \\ y_{n+1} &= y_n + h \frac{s_1 + s_2}{2}, \\ t_{n+1} &= t_n + h. \end{aligned}$$

If we were to use both of these methods simultaneously, they would produce two different values for y_{n+1}. The difference between the two values would provide an error estimate and a basis for picking the step size. Furthermore, an extrapolated combination of the two values would be more accurate than either one individually.

Continuing with this approach is the idea behind *single-step* methods for integrating ordinary differential equations. The function $f(t, y)$ is evaluated several times for values of t between t_n and t_{n+1} and values of y obtained by adding linear combinations of the values of f to y_n. The actual step is taken using another linear combination of the function values. Modern versions of single-step methods use yet another linear combination of function values to estimate error and determine step size.

Single-step methods are often called *Runge–Kutta* methods, after the two German applied mathematicians who first wrote about them around 1905. The classical Runge–Kutta method was widely used for hand computation before the invention of digital computers and is still popular today. It uses four function evaluations per

step:

$$
\begin{aligned}
s_1 &= f(t_n, y_n), \\
s_2 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}s_1\right), \\
s_3 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}s_2\right), \\
s_4 &= f(t_n + h, y_n + hs_3), \\
y_{n+1} &= y_n + \frac{h}{6}(s_1 + 2s_2 + 2s_3 + s_4), \\
t_{n+1} &= t_n + h.
\end{aligned}
$$

If $f(t, y)$ does not depend on y, then classical Runge–Kutta has $s_2 = s_3$ and the method reduces to Simpson's quadrature rule.

Classical Runge–Kutta does not provide an error estimate. The method is sometimes used with a step size h and again with step size $h/2$ to obtain an error estimate, but we now know more efficient methods.

Several of the ordinary differential equation solvers in MATLAB, including the textbook solver we describe later in this chapter, are single-step or Runge–Kutta solvers. A general single-step method is characterized by a number of parameters, α_i, $\beta_{i,j}$, γ_i, and δ_i. There are k *stages*. Each stage computes a slope, s_i, by evaluating $f(t, y)$ for a particular value of t and a value of y obtained by taking linear combinations of the previous slopes:

$$
s_i = f\left(t_n + \alpha_i h, y_n + h\sum_{j=1}^{i-1} \beta_{i,j} s_j\right), \ i = 1, \ldots, k.
$$

The proposed step is also a linear combination of the slopes:

$$
y_{n+1} = y_n + h\sum_{i=1}^{k} \gamma_i s_i.
$$

An estimate of the error that would occur with this step is provided by yet another linear combination of the slopes:

$$
e_{n+1} = h\sum_{i=1}^{k} \delta_i s_i.
$$

If this error is less than the specified tolerance, then the step is successful and y_{n+1} is accepted. If not, the step is a failure and y_{n+1} is rejected. In either case, the error estimate is used to compute the step size h for the next step.

The parameters in these methods are determined by matching terms in Taylor series expansions of the slopes. These series involve powers of h and products of various partial derivatives of $f(t, y)$. The *order* of a method is the exponent of the smallest power of h that cannot be matched. It turns out that one, two, three, and

four stages yield methods of order one, two, three, and four, respectively. But it takes six stages to obtain a fifth-order method. The classical Runge–Kutta method has four stages and is fourth order.

The names of the MATLAB ordinary differential equation solvers are all of the form `odennxx` with digits `nn` indicating the order of the underlying method and a possibly empty `xx` indicating some special characteristic of the method. If the error estimate is obtained by comparing formulas with different orders, the digits `nn` indicate these orders. For example, `ode45` obtains its error estimate by comparing a fourth-order and a fifth-order formula.

7.5 The BS23 Algorithm

Our textbook function `ode23tx` is a simplified version of the function `ode23` that is included with MATLAB. The algorithm is due to Bogacki and Shampine [3, 6]. The "23" in the function names indicates that two simultaneous single-step formulas, one of second order and one of third order, are involved.

The method has three stages, but there are four slopes s_i because, after the first step, the s_1 for one step is the s_4 from the previous step. The essentials are

$$\begin{aligned}
s_1 &= f(t_n, y_n), \\
s_2 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}s_1\right), \\
s_3 &= f\left(t_n + \frac{3}{4}h, y_n + \frac{3}{4}hs_2\right), \\
t_{n+1} &= t_n + h, \\
y_{n+1} &= y_n + \frac{h}{9}(2s_1 + 3s_2 + 4s_3), \\
s_4 &= f(t_{n+1}, y_{n+1}), \\
e_{n+1} &= \frac{h}{72}(-5s_1 + 6s_2 + 8s_3 - 9s_4).
\end{aligned}$$

The simplified pictures in Figure 7.1 show the starting situation and the three stages. We start at a point (t_n, y_n) with an initial slope $s_1 = f(t_n, y_n)$ and an estimate of a good step size, h. Our goal is to compute an approximate solution y_{n+1} at $t_{n+1} = t_n + h$ that agrees with the true solution $y(t_{n+1})$ to within the specified tolerances.

The first stage uses the initial slope s_1 to take an Euler step halfway across the interval. The function is evaluated there to get the second slope, s_2. This slope is used to take an Euler step three-quarters of the way across the interval. The function is evaluated again to get the third slope, s_3. A weighted average of the three slopes,

$$s = \frac{1}{9}(2s_1 + 3s_2 + 4s_3),$$

is used for the final step all the way across the interval to get a tentative value for y_{n+1}. The function is evaluated once more to get s_4. The error estimate then uses

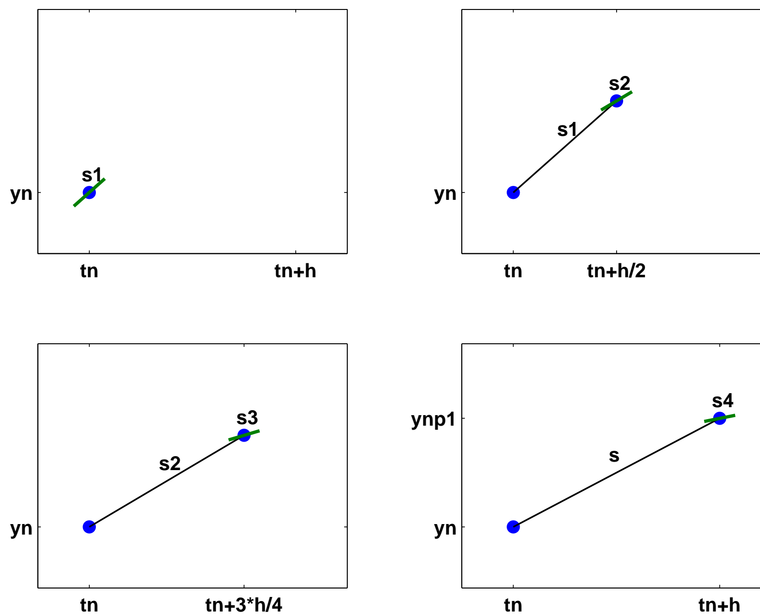

Figure 7.1. *BS23 algorithm.*

all four slopes:

$$e_{n+1} = \frac{h}{72}(-5s_1 + 6s_2 + 8s_3 - 9s_4).$$

If the error is within the specified tolerance, then the step is successful, the tentative value of y_{n+1} is accepted, and s_4 becomes the s_1 of the next step. If the error is too large, then the tentative y_{n+1} is rejected and the step must be redone. In either case, the error estimate e_{n+1} provides the basis for determining the step size h for the next step.

The first input argument of `ode23tx` specifies the function $f(t, y)$. This argument can take any of three different forms:

- a function handle,
- an inline function, or
- an anonymous function (MATLAB 7).

The function should accept two arguments—usually, but not necessarily, `t` and `y`. The result of evaluating the character string or the function should be a column vector containing the values of the derivatives, dy/dt.

The second input argument of `ode23tx` is a vector, `tspan`, with two components, `t0` and `tfinal`. The integration is carried out over the interval

$$t_0 \le t \le t_{final}.$$

One of the simplifications in our textbook code is this form of tspan. Other MATLAB ordinary differential equation solvers allow more flexible specifications of the integration interval.

The third input argument is a column vector, y0, providing the initial value of $y_0 = y(t_0)$. The length of y0 tells ode23tx the number of differential equations in the system.

A fourth input argument is optional and can take two different forms. The simplest, and most common, form is a scalar numerical value, rtol, to be used as the relative error tolerance. The default value for rtol is 10^{-3}, but you can provide a different value if you want more or less accuracy. The more complicated possibility for this optional argument is the structure generated by the MATLAB function odeset. This function takes pairs of arguments that specify many different options for the MATLAB ordinary differential equation solvers. For ode23tx, you can change the default values of three quantities: the relative error tolerance, the absolute error tolerance, and the M-file that is called after each successful step. The statement

```
opts = odeset('reltol',1.e-5, 'abstol',1.e-8, ...
              'outputfcn',@myodeplot)
```

creates a structure that specifies the relative error tolerance to be 10^{-5}, the absolute error tolerance to be 10^{-8}, and the output function to be myodeplot.

The output produced by ode23tx can be either graphic or numeric. With no output arguments, the statement

```
ode23tx(F,tspan,y0);
```

produces a dynamic plot of all the components of the solution. With two output arguments, the statement

```
[tout,yout] = ode23tx(F,tspan,y0);
```

generates a table of values of the solution.

7.6 ode23tx

Let's examine the code for ode23tx. Here is the preamble.

```
function [tout,yout] = ode23tx(F,tspan,y0,arg4,varargin)
%ODE23TX  Solve non-stiff differential equations.
%         Textbook version of ODE23.
%
%   ODE23TX(F,TSPAN,Y0) with TSPAN = [T0 TFINAL]
%   integrates the system of differential equations
%   dy/dt = f(t,y) from t = T0 to t = TFINAL.
%   The initial condition is y(T0) = Y0.
%
%   The first argument, F, is a function handle, an
```

```
%   inline object in MATLAB6, or an anonymous function
%   in MATLAB7, that defines f(t,y).  This function
%   must have two input arguments, t and y, and must
%   return a column vector of the derivatives, dy/dt.
%
%   With two output arguments, [T,Y] = ODE23TX(...)
%   returns a column vector T and an array Y where Y(:,k)
%   is the solution at T(k).
%
%   With no output arguments, ODE23TX plots the solution.
%
%   ODE23TX(F,TSPAN,Y0,RTOL) uses the relative error
%   tolerance RTOL instead of the default 1.e-3.
%
%   ODE23TX(F,TSPAN,Y0,OPTS) where OPTS = ...
%   ODESET('reltol',RTOL,'abstol',ATOL,'outputfcn',@PLTFN)
%   uses relative error RTOL instead of 1.e-3,
%   absolute error ATOL instead of 1.e-6, and calls PLTFN
%   instead of ODEPLOT after each step.
%
%   More than four input arguments, ODE23TX(F,TSPAN,Y0,
%   RTOL,P1,P2,..), are passed on to F, F(T,Y,P1,P2,..).
%
%   ODE23TX uses the Runge-Kutta (2,3) method of
%   Bogacki and Shampine.
%
%   Example
%      tspan = [0 2*pi];
%      y0 = [1 0]';
%      F = '[0 1; -1 0]*y';
%      ode23tx(F,tspan,y0);
%
%   See also ODE23.
```

Here is the code that parses the arguments and initializes the internal variables.

```
rtol = 1.e-3;
atol = 1.e-6;
plotfun = @odeplot;
if nargin >= 4 & isnumeric(arg4)
   rtol = arg4;
elseif nargin >= 4 & isstruct(arg4)
   if ~isempty(arg4.RelTol), rtol = arg4.RelTol; end
   if ~isempty(arg4.AbsTol), atol = arg4.AbsTol; end
   if ~isempty(arg4.OutputFcn),
         plotfun = arg4.OutputFcn; end
end
```

```
t0 = tspan(1);
tfinal = tspan(2);
tdir = sign(tfinal - t0);
plotit = (nargout == 0);
threshold = atol / rtol;
hmax = abs(0.1*(tfinal-t0));
t = t0;
y = y0(:);

% Initialize output.

if plotit
   feval(plotfun,tspan,y,'init');
else
   tout = t;
   yout = y.';
end
```

The computation of the initial step size is a delicate matter because it requires some knowledge of the overall scale of the problem.

```
s1 = feval(F, t, y, varargin{:});
r = norm(s1./max(abs(y),threshold),inf) + realmin;
h = tdir*0.8*rtol^(1/3)/r;
```

Here is the beginning of the main loop. The integration starts at $t = t_0$ and increments t until it reaches t_{final}. It is possible to go "backward," that is, have $t_{final} < t_0$.

```
while t ~= tfinal

   hmin = 16*eps*abs(t);
   if abs(h) > hmax, h = tdir*hmax; end
   if abs(h) < hmin, h = tdir*hmin; end

   % Stretch the step if t is close to tfinal.

   if 1.1*abs(h) >= abs(tfinal - t)
      h = tfinal - t;
   end
```

Here is the actual computation. The first slope `s1` has already been computed. The function defining the differential equation is evaluated three more times to obtain three more slopes.

```
   s2 = feval(F, t+h/2, y+h/2*s1, varargin{:});
   s3 = feval(F, t+3*h/4, y+3*h/4*s2, varargin{:});
   tnew = t + h;
```

```
    ynew = y + h*(2*s1 + 3*s2 + 4*s3)/9;
    s4 = feval(F, tnew, ynew, varargin{:});
```

Here is the error estimate. The norm of the error vector is scaled by the ratio of the absolute tolerance to the relative tolerance. The use of the smallest floating-point number, `realmin`, prevents `err` from being exactly zero.

```
    e = h*(-5*s1 + 6*s2 + 8*s3 - 9*s4)/72;
    err = norm(e./max(max(abs(y),abs(ynew)),threshold),
          ... inf) + realmin;
```

Here is the test to see if the step is successful. If it is, the result is plotted or appended to the output vector. If it is not, the result is simply forgotten.

```
    if err <= rtol
      t = tnew;
      y = ynew;
      if plotit
         if feval(plotfun,t,y,'');
            break
         end
      else
         tout(end+1,1) = t;
         yout(end+1,:) = y.';
      end
      s1 = s4; % Reuse final function value to start new step.
    end
```

The error estimate is used to compute a new step size. The ratio `rtol/err` is greater than one if the current step is successful, or less than one if the current step fails. A cube root is involved because the BS23 is a third-order method. This means that changing tolerances by a factor of eight will change the typical step size, and hence the total number of steps, by a factor of two. The factors 0.8 and 5 prevent excessive changes in step size.

```
    % Compute a new step size.
    h = h*min(5,0.8*(rtol/err)^(1/3));
```

Here is the only place where a singularity would be detected.

```
    if abs(h) <= hmin
       warning(sprintf( ...
          'Step size %e too small at t = %e.\n',h,t));
       t = tfinal;
    end
 end
```

That ends the main loop. The plot function might need to finish its work.

```
 if plotit
    feval(plotfun,[],[],'done');
 end
```

7.7 Examples

Please sit down in front of a computer running MATLAB. Make sure `ode23tx` is in your current directory or on your MATLAB path. Start your session by entering

```
F = inline('0','t','y');  ode23tx(F,[0 10],1)
```

or

```
F = @(t,x) 0 ;  ode23tx(F,[0 10],1)
```

This should produce a plot of the solution of the initial value problem

$$\frac{dy}{dt} = 0,$$
$$y(0) = 1,$$
$$0 \le t \le 10.$$

The solution, of course, is a constant function, $y(t) = 1$.

Now you can press the up arrow key, use the left arrow key to space over to the `0`, and change it to something more interesting. Here are some examples. At first, we'll change just the `0` and leave the `[0 10]` and `1` alone.

```
F                    Exact solution
0                    1
t                    1+t^2/2
y                    exp(t)
-y                   exp(-t)
1/(1-3*t)            1-log(1-3*t)/3   (Singular)
2*y-y^2              2/(1+exp(-2*t))
```

Make up some of your own examples. Change the initial condition. Change the accuracy by including `1.e-6` as the fourth argument.

Now let's try the harmonic oscillator, a second-order differential equation written as a pair of two first-order equations. First, create an inline function to specify the equations. In MATLAB 6 use either

```
F = inline('[y(2); -y(1)]','t','y')
```

or

```
F = inline('[0 1; -1 0]*y','t','y')
```

In MATLAB 7 use either

```
F = @(t,y) [y(2); -y(1)];
```

or

```
F = @(t,y) [0 1; -1 0]*y;
```

Then the statement

```
ode23tx(F,[0 2*pi],[1; 0])
```

plots two functions of t that you should recognize. If you want to produce a *phase plane* plot, you have two choices. One possibility is to capture the output and plot it after the computation is complete.

```
[t,y] = ode23tx(F,[0 2*pi],[1; 0])
plot(y(:,1),y(:,2),'-o')
axis([-1.2 1.2 -1.2 1.2])
axis square
```

The more interesting possibility is to use a function that plots the solution while it is being computed. MATLAB provides such a function in `odephas2.m`. It is accessed by using `odeset` to create an options structure.

```
opts = odeset('reltol',1.e-4,'abstol',1.e-6, ...
   'outputfcn',@odephas2);
```

If you want to provide your own plotting function, it should be something like

```
function flag = phaseplot(t,y,job)
persistent p
if isequal(job,'init')
   p = plot(y(1),y(2),'o','erasemode','none');
   axis([-1.2 1.2 -1.2 1.2])
   axis square
   flag = 0;
elseif isequal(job,'')
   set(p,'xdata',y(1),'ydata',y(2))
   drawnow
   flag = 0;
end
```

This is with

```
opts = odeset('reltol',1.e-4,'abstol',1.e-6, ...
   'outputfcn',@phaseplot);
```

Once you have decided on a plotting function and created an options structure, you can compute and simultaneously plot the solution with

```
ode23tx(F,[0 2*pi],[1; 0],opts)
```

Try this with other values of the tolerances.

Issue the command `type twobody` to see if there is an M-file `twobody.m` on your path. If not, find the two or three lines of code earlier in this chapter and create your own M-file. Then try

```
ode23tx(@twobody,[0 2*pi],[1; 0; 0; 1]);
```

The code, and the length of the initial condition, indicate that the solution has four components. But the plot shows only three. Why? Hint: Find the `zoom` button on the figure window toolbar and zoom in on the blue curve.

You can vary the initial condition of the two-body problem by changing the fourth component.

```
y0 = [1; 0; 0; change_this];
ode23tx(@twobody,[0 2*pi],y0);
```

Graph the orbit, and the heavy body at the origin, with

```
y0 = [1; 0; 0; change_this];
[t,y] = ode23tx(@twobody,[0 2*pi],y0);
plot(y(:,1),y(:,2),'-',0,0,'ro')
axis equal
```

You might also want to use something other than 2π for `tfinal`.

7.8 Lorenz Attractor

One of the world's most extensively studied ordinary differential equations is the Lorenz chaotic attractor. It was first described in 1963 by Edward Lorenz, an M.I.T. mathematician and meteorologist who was interested in fluid flow models of the earth's atmosphere. An excellent reference is a book by Colin Sparrow [8].

We have chosen to express the Lorenz equations in a somewhat unusual way involving a matrix-vector product:

$$\dot{y} = Ay.$$

The vector y has three components that are functions of t:

$$y(t) = \begin{pmatrix} y_1(t) \\ y_2(t) \\ y_3(t) \end{pmatrix}.$$

Despite the way we have written it, this is not a linear system of differential equations. Seven of the nine elements in the 3-by-3 matrix A are constant, but the other two depend on $y_2(t)$:

$$A = \begin{bmatrix} -\beta & 0 & y_2 \\ 0 & -\sigma & \sigma \\ -y_2 & \rho & -1 \end{bmatrix}.$$

The first component of the solution, $y_1(t)$, is related to the convection in the atmospheric flow, while the other two components are related to horizontal and vertical temperature variation. The parameter σ is the Prandtl number, ρ is the normalized Rayleigh number, and β depends on the geometry of the domain. The most popular values of the parameters, $\sigma = 10$, $\rho = 28$, and $\beta = 8/3$, are outside the ranges associated with the earth's atmosphere.

The deceptively simple nonlinearity introduced by the presence of y_2 in the system matrix A changes everything. There are no random aspects to these equations, so the solutions $y(t)$ are completely determined by the parameters and the initial conditions, but their behavior is very difficult to predict. For some values of the parameters, the orbit of $y(t)$ in three-dimensional space is known as a *strange attractor*. It is bounded, but not periodic and not convergent. It never intersects itself. It ranges chaotically back and forth around two different points, or attractors. For other values of the parameters, the solution might converge to a fixed point, diverge to infinity, or oscillate periodically. See Figures 7.2 and 7.3.

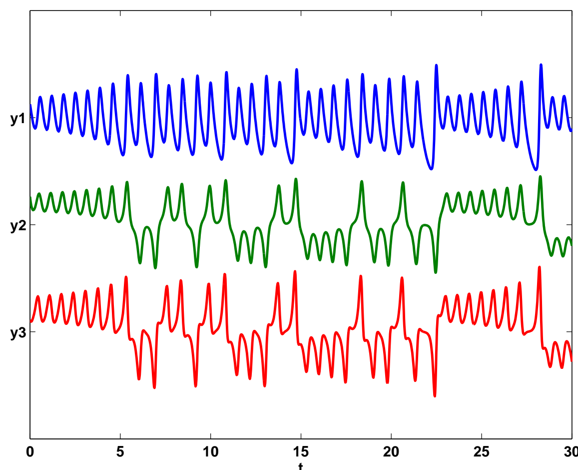

Figure 7.2. *Three components of Lorenz attractor.*

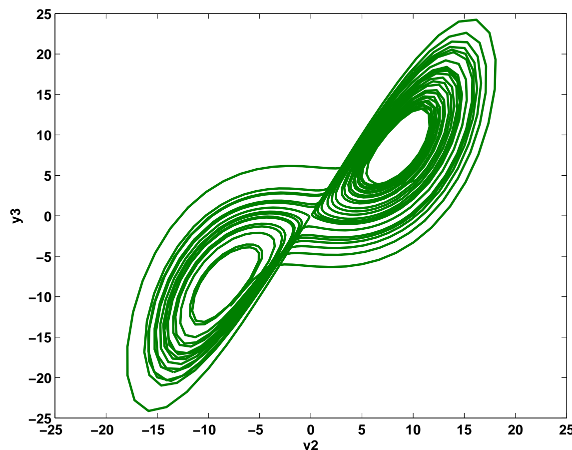

Figure 7.3. *Phase plane plot of Lorenz attractor.*

Let's think of $\eta = y_2$ as a free parameter, restrict ρ to be greater than one, and study the matrix

$$A = \begin{bmatrix} -\beta & 0 & \eta \\ 0 & -\sigma & \sigma \\ -\eta & \rho & -1 \end{bmatrix}.$$

It turns out that A is singular if and only if

$$\eta = \pm\sqrt{\beta(\rho-1)}.$$

The corresponding null vector, normalized so that its second component is equal to η, is

$$\begin{pmatrix} \rho-1 \\ \eta \\ \eta \end{pmatrix}.$$

With two different signs for η, this defines two points in three-dimensional space. These points are fixed points for the differential equation. If

$$y(t_0) = \begin{pmatrix} \rho-1 \\ \eta \\ \eta \end{pmatrix},$$

then, for all t,

$$\dot{y}(t) = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix},$$

and so $y(t)$ never changes. However, these points are unstable fixed points. If $y(t)$ does not start at one of these points, it will never reach either of them; if it tries to approach either point, it will be repulsed.

We have provided an M-file, `lorenzgui.m`, that facilitates experiments with the Lorenz equations. Two of the parameters, $\beta = 8/3$ and $\sigma = 10$, are fixed. A `uicontrol` offers a choice among several different values of the third parameter, ρ. A simplified version of the program for $\rho = 28$ would begin with

```
rho = 28;
sigma = 10;
beta = 8/3;
eta = sqrt(beta*(rho-1));
A = [ -beta    0     eta
          0  -sigma   sigma
       -eta   rho    -1  ];
```

The initial condition is taken to be near one of the attractors.

```
yc = [rho-1; eta; eta];
y0 = yc + [0; 0; 3];
```

The time span is infinite, so the integration will have to be stopped by another `uicontrol`.

```
tspan = [0 Inf];
opts = odeset('reltol',1.e-6,'outputfcn',@lorenzplot);
ode45(@lorenzeqn, tspan, y0, opts, A);
```

The matrix A is passed as an extra parameter to the integrator, which sends it on to `lorenzeqn`, the subfunction defining the differential equation. The extra parameter machinery included in the function functions allows `lorenzeqn` to be written in a particularly compact manner.

```
function ydot = lorenzeqn(t,y,A)
A(1,3) = y(2);
A(3,1) = -y(2);
ydot = A*y;
```

Most of the complexity of `lorenzgui` is contained in the plotting subfunction, `lorenzplot`. It not only manages the user interface controls, it must also anticipate the possible range of the solution in order to provide appropriate axis scaling.

7.9 Stiffness

Stiffness is a subtle, difficult, and important concept in the numerical solution of ordinary differential equations. It depends on the differential equation, the initial conditions, and the numerical method. Dictionary definitions of the word "stiff" involve terms like "not easily bent," "rigid," and "stubborn." We are concerned with a computational version of these properties.

> *A problem is stiff if the solution being sought varies slowly, but there are nearby solutions that vary rapidly, so the numerical method must take small steps to obtain satisfactory results.*

Stiffness is an efficiency issue. If we weren't concerned with how much time a computation takes, we wouldn't be concerned about stiffness. Nonstiff methods can solve stiff problems; they just take a long time to do it.

A model of flame propagation provides an example. We learned about this example from Larry Shampine, one of the authors of the MATLAB ordinary differential equation suite. If you light a match, the ball of flame grows rapidly until it reaches a critical size. Then it remains at that size because the amount of oxygen being consumed by the combustion in the interior of the ball balances the amount available through the surface. The simple model is

$$\dot{y} = y^2 - y^3,$$
$$y(0) = \delta,$$
$$0 \le t \le 2/\delta.$$

The scalar variable $y(t)$ represents the radius of the ball. The y^2 and y^3 terms come from the surface area and the volume. The critical parameter is the initial radius,

δ, which is "small." We seek the solution over a length of time that is inversely proportional to δ.

At this point, we suggest that you start up MATLAB and actually run our examples. It is worthwhile to see them in action. We will start with `ode45`, the workhorse of the MATLAB ordinary differential equation suite. If δ is not very small, the problem is not very stiff. Try $\delta = 0.01$ and request a relative error of 10^{-4}.

```
delta = 0.01;
F = inline('y^2 - y^3','t','y');
opts = odeset('RelTol',1.e-4);
ode45(F,[0 2/delta],delta,opts);
```

With no output arguments, `ode45` automatically plots the solution as it is computed. You should get a plot of a solution that starts at $y = 0.01$, grows at a modestly increasing rate until t approaches 100, which is $1/\delta$, then grows rapidly until it reaches a value close to 1, where it remains.

Now let's see stiffness in action. Decrease δ by a couple of orders of magnitude. (If you run only one example, run this one.)

```
delta = 0.0001;
ode45(F,[0 2/delta],delta,opts);
```

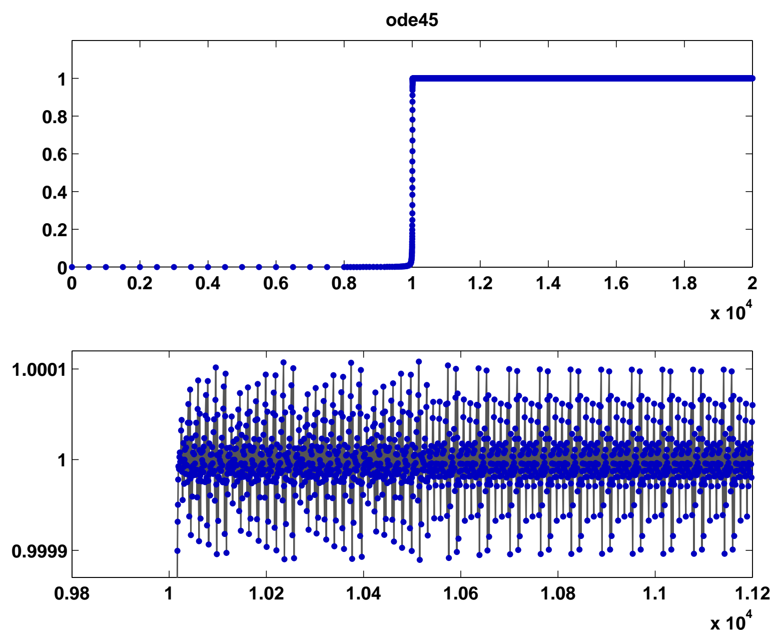

Figure 7.4. *Stiff behavior of* `ode45`.

You should see something like Figure 7.4, although it will take a long time to complete the plot. If you get tired of watching the agonizing progress, click

the stop button in the lower left corner of the window. Turn on zoom, and use the mouse to explore the solution near where it first approaches steady state. You should see something like the detail in Figure 7.4. Notice that `ode45` is doing its job. It's keeping the solution within 10^{-4} of its nearly constant steady state value. But it certainly has to work hard to do it. If you want an even more dramatic demonstration of stiffness, decrease the tolerance to 10^{-5} or 10^{-6}.

This problem is not stiff initially. It only becomes stiff as the solution approaches steady state. This is because the steady state solution is so "rigid." Any solution near $y(t) = 1$ increases or decreases rapidly toward that solution. (We should point out that "rapidly" here is with respect to an unusually long time scale.)

What can be done about stiff problems? You don't want to change the differential equation or the initial conditions, so you have to change the numerical method. Methods intended to solve stiff problems efficiently do more work per step, but can take much bigger steps. Stiff methods are *implicit*. At each step they use MATLAB matrix operations to solve a system of simultaneous linear equations that helps predict the evolution of the solution. For our flame example, the matrix is only 1 by 1, but even here, stiff methods do more work per step than nonstiff methods.

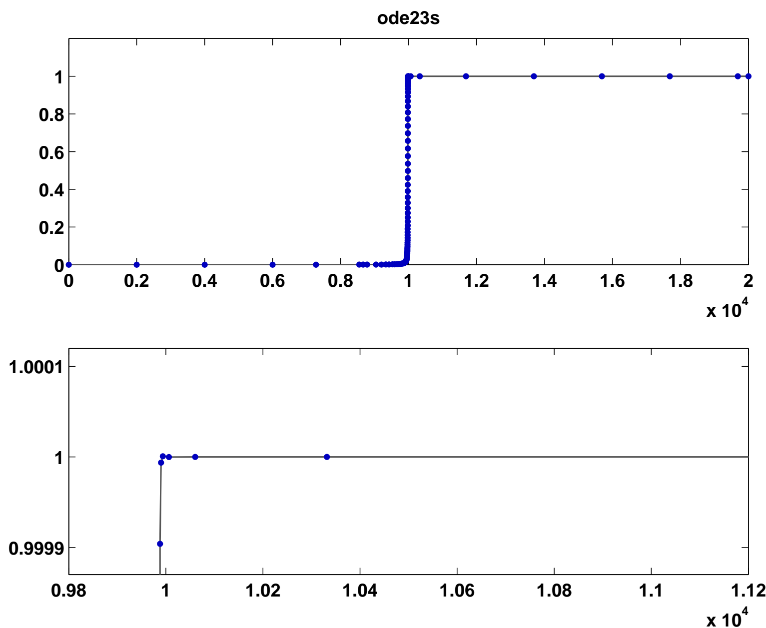

Figure 7.5. *Stiff behavior of* `ode23s`.

Let's compute the solution to our flame example again, this time with one of the ordinary differential equation solvers in MATLAB whose name ends in "`s`" for "stiff."

```
delta = 0.0001;
ode23s(F,[0 2/delta],delta,opts);
```

Figure 7.5 shows the computed solution and the zoom detail. You can see that `ode23s` takes many fewer steps than `ode45`. This is actually an easy problem for a stiff solver. In fact, `ode23s` takes only 99 steps and uses just 412 function evaluations, while `ode45` takes 3040 steps and uses 20179 function evaluations. Stiffness even affects graphical output. The print files for the `ode45` figures are much larger than those for the `ode23s` figures.

Imagine you are returning from a hike in the mountains. You are in a narrow canyon with steep slopes on either side. An explicit algorithm would sample the local gradient to find the descent direction. But following the gradient on either side of the trail will send you bouncing back and forth across the canyon, as with `ode45`. You will eventually get home, but it will be long after dark before you arrive. An implicit algorithm would have you keep your eyes on the trail and anticipate where each step is taking you. It is well worth the extra concentration.

This flame problem is also interesting because it involves the Lambert W function, $W(z)$. The differential equation is separable. Integrating once gives an implicit equation for y as a function of t:

$$\frac{1}{y} + \log\left(\frac{1}{y} - 1\right) = \frac{1}{\delta} + \log\left(\frac{1}{\delta} - 1\right) - t.$$

This equation can be solved for y. The exact analytical solution to the flame model turns out to be

$$y(t) = \frac{1}{W(ae^{a-t}) + 1},$$

where $a = 1/\delta - 1$. The function $W(z)$, the Lambert W function, is the solution to

$$W(z)e^{W(z)} = z.$$

With MATLAB and the Symbolic Math Toolbox connection to Maple, the statements

```
y = dsolve('Dy = y^2 - y^3','y(0) = 1/100');
y = simplify(y);
pretty(y)
ezplot(y,0,200)
```

produce

```
                1
    ----------------------------
    lambertw(99 exp(99 - t)) + 1
```

and the plot of the exact solution shown in Figure 7.6. If the initial value 1/100 is decreased and the time span $0 \leq t \leq 200$ increased, the transition region becomes narrower.

The Lambert W function is named after J. H. Lambert (1728–1777). Lambert was a colleague of Euler and Lagrange's at the Berlin Academy of Sciences and is best known for his laws of illumination and his proof that π is irrational. The function was "rediscovered" a few years ago by Corless, Gonnet, Hare, and Jeffrey, working on Maple, and by Don Knuth [4].

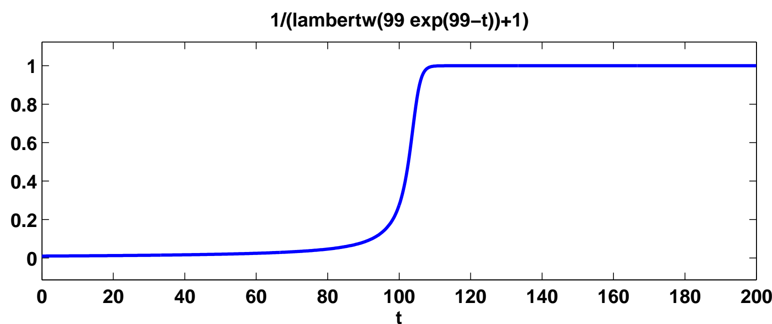

Figure 7.6. *Exact solution for the flame example.*

7.10 Events

So far, we have been assuming that the `tspan` interval, $t_0 \le t \le t_{final}$, is a given part of the problem specification, or we have used an infinite interval and a GUI button to terminate the computation. In many situations, the determination of t_{final} is an important aspect of the problem.

One example is a body falling under the force of gravity and encountering air resistance. When does it hit the ground? Another example is the two-body problem, the orbit of one body under the gravitational attraction of a much heavier body. What is the period of the orbit? The *events* feature of the MATLAB ordinary differential equation solvers provides answers to such questions.

Events detection in ordinary differential equations involves two functions, $f(t,y)$ and $g(t,y)$, and an initial condition, (t_0, y_0). The problem is to find a function $y(t)$ and a final value t_* so that

$$\dot{y} = f(t,y),$$

$$y(t_0) = y_0,$$

and

$$g(t_*, y(t_*)) = 0.$$

A simple model for the falling body is

$$\ddot{y} = -1 + \dot{y}^2,$$

with initial conditions $y(0) = 1$, $\dot{y}(0) = 0$. The question is, for what t does $y(t) = 0$? The code for the function $f(t,y)$ is

```
function ydot = f(t,y)
ydot = [y(2); -1+y(2)^2];
```

With the differential equation written as a first-order system, y becomes a vector with two components and so $g(t,y) = y_1$. The code for $g(t,y)$ is

```
function [gstop,isterminal,direction] = g(t,y)
gstop = y(1);
isterminal = 1;
direction = [];
```

The first output, gstop, is the value that we want to make zero. Setting the second output, isterminal, to one indicates that the ordinary differential equation solver should terminate when gstop is zero. Setting the third output, direction, to the empty matrix indicates that the zero can be approached from either direction. With these two functions available, the following statements compute and plot the trajectory shown in Figure 7.7.

```
opts = odeset('events',@g);
y0 = [1; 0];
[t,y,tfinal] = ode45(@f,[0 Inf],y0,opts);
tfinal
plot(t,y(:,1),'-',[0 tfinal],[1 0],'o')
axis([-.1 tfinal+.1 -.1 1.1])
xlabel('t')
ylabel('y')
title('Falling body')
text(1.2, 0, ['tfinal = ' num2str(tfinal)])
```

The terminating value of t is found to be tfinal = 1.6585.

The three sections of code for this example can be saved in three separate M-files, with two functions and one script, or they can all be saved in one function M-file. In the latter case, f and g become subfunctions and have to appear after the main body of code.

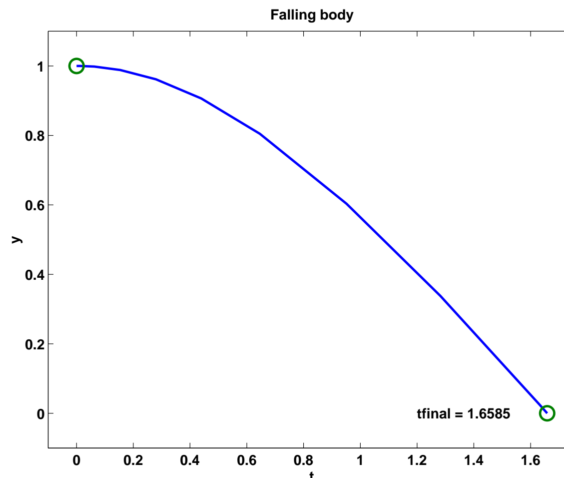

Figure 7.7. *Event handling for falling object.*

Events detection is particularly useful in problems involving periodic phenomena. The two-body problem provides a good example. Here is the first portion of a function M-file, `orbit.m`. The input parameter is `reltol`, the desired local relative tolerance.

```
function orbit(reltol)
y0 = [1; 0; 0; 0.3];
opts = odeset('events',@gstop,'reltol',reltol);
[t,y,te,ye] = ode45(@twobody,[0 2*pi],y0,opts,y0);
tfinal = te(end)
yfinal = ye(end,1:2)
plot(y(:,1),y(:,2),'-',0,0,'ro')
axis([-.1 1.05 -.35 .35])
```

The function `ode45` is used to compute the orbit. The first input argument is a function handle, `@twobody`, that references the function defining the differential equations. The second argument to `ode45` is any overestimate of the time interval required to complete one period. The third input argument is `y0`, a 4-vector that provides the initial position and velocity. The light body starts at $(1, 0)$, which is a point with a distance 1 from the heavy body, and has initial velocity $(0, 0.3)$, which is perpendicular to the initial position vector. The fourth input argument is an options structure created by `odeset` that overrides the default value for `reltol` and that specifies a function `gstop` that defines the events we want to locate. The last argument is `y0`, an "extra" argument that `ode45` passes on to both `twobody` and `gstop`.

The code for `twobody` has to be modified to accept a third argument, even though it is not used.

```
function ydot = twobody(t,y,y0)
r = sqrt(y(1)^2 + y(2)^2);
ydot = [y(3); y(4); -y(1)/r^3; -y(2)/r^3];
```

The ordinary differential equation solver calls the `gstop` function at every step during the integration. This function tells the solver whether or not it is time to stop.

```
function [val,isterm,dir] = gstop(t,y,y0)
d = y(1:2)-y0(1:2);
v = y(3:4);
val = d'*v;
isterm = 1;
dir = 1;
```

The 2-vector `d` is the difference between the current position and the starting point. The 2-vector `v` is the velocity at the current position. The quantity `val` is the inner product between these two vectors. Mathematically, the stopping function is

$$g(t, y) = \dot{d}(t)^T d(t),$$

where

$$d = (y_1(t) - y_1(0), y_2(t) - y_2(0))^T.$$

Points where $g(t, y(t)) = 0$ are the local minimum or maximum of $d(t)$. By setting `dir = 1`, we indicate that the zeros of $g(t, y)$ must be approached from above, so they correspond to minima. By setting `isterm = 1`, we indicate that computation of the solution should be terminated at the first minimum. If the orbit is truly periodic, then any minima of d occur when the body returns to its starting point.

Calling `orbit` with a very loose tolerance

```
orbit(2.0e-3)
```

produces

```
tfinal =
   2.35087197761898

yfinal =
   0.98107659901079  -0.00012519138559
```

and plots Figure 7.8.

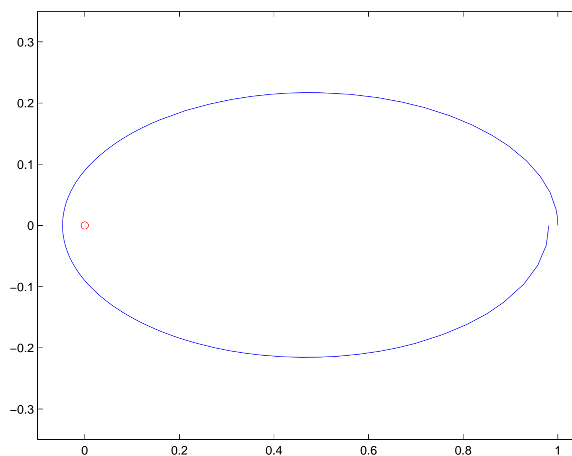

Figure 7.8. *Periodic orbit computed with loose tolerance.*

You can see from both the value of `yfinal` and the graph that the orbit does not quite return to the starting point. We need to request more accuracy.

```
orbit(1.0e-6)
```

produces

```
tfinal =
   2.38025846171805

yfinal =
   0.99998593905521   0.00000000032240
```

Now the value of `yfinal` is close enough to `y0` that the graph of the orbit is effectively closed.

7.11 Multistep Methods

A single-step numerical method has a short memory. The only information passed from one step to the next is an estimate of the proper step size and, perhaps, the value of $f(t_n, y_n)$ at the point the two steps have in common.

As the name implies, a multistep method has a longer memory. After an initial start-up phase, a pth-order multistep method saves up to perhaps a dozen values of the solution, $y_{n-p+1}, y_{n-p+2}, \ldots, y_{n-1}, y_n$, and uses them all to compute y_{n+1}. In fact, these methods can vary both the order, p, and the step size, h.

Multistep methods tend to be more efficient than single-step methods for problems with smooth solutions and high accuracy requirements. For example, the orbits of planets and deep space probes are computed with multistep methods.

7.12 The Matlab ODE Solvers

This section is derived from the Algorithms portion of the Matlab Reference Manual page for the ordinary differential equation solvers.

`ode45` is based on an explicit Runge–Kutta (4, 5) formula, the Dormand–Prince pair. It is a one-step solver. In computing $y(t_{n+1})$, it needs only the solution at the immediately preceding time point, $y(t_n)$. In general, `ode45` is the first function to try for most problems.

`ode23` is an implementation of an explicit Runge–Kutta (2, 3) pair of Bogacki and Shampine's. It is often more efficient than `ode45` at crude tolerances and in the presence of moderate stiffness. Like `ode45`, `ode23` is a one-step solver.

`ode113` uses a variable-order Adams–Bashforth–Moulton predictor-corrector algorithm. It is often more efficient than `ode45` at stringent tolerances and if the ordinary differential equation file function is particularly expensive to evaluate. `ode113` is a multistep solver—it normally needs the solutions at several preceding time points to compute the current solution.

The above algorithms are intended to solve nonstiff systems. If they appear to be unduly slow, try using one of the stiff solvers below.

`ode15s` is a variable-order solver based on the numerical differentiation formulas (NDFs). Optionally, it uses the backward differentiation formulas (BDFs, also known as Gear's method), which are usually less efficient. Like `ode113`, `ode15s` is a multistep solver. Try `ode15s` if `ode45` fails or is very inefficient and you suspect that the problem is stiff, or if you are solving a differential-algebraic problem.

`ode23s` is based on a modified Rosenbrock formula of order two. Because it is a one-step solver, it is often more efficient than `ode15s` at crude tolerances. It can solve some kinds of stiff problems for which `ode15s` is not effective.

`ode23t` is an implementation of the trapezoidal rule using a "free" interpolant. Use this solver if the problem is only moderately stiff and you need a solution without numerical damping. `ode23t` can solve differential-algebraic equations.

`ode23tb` is an implementation of TR-BDF2, an implicit Runge–Kutta formula with a first stage that is a trapezoidal rule step and a second stage that is a BDF of order two. By construction, the same iteration matrix is used in evaluating both stages. Like `ode23s`, this solver is often more efficient than `ode15s` at crude tolerances.

Here is a summary table from the MATLAB Reference Manual. For each function, it lists the appropriate problem type, the typical accuracy of the method, and the recommended area of usage.

- `ode45`. Nonstiff problems, medium accuracy. Use most of the time. This should be the first solver you try.

- `ode23`. Nonstiff problems, low accuracy. Use for large error tolerances or moderately stiff problems.

- `ode113`. Nonstiff problems, low to high accuracy. Use for stringent error tolerances or computationally intensive ordinary differential equation functions.

- `ode15s`. Stiff problems, low to medium accuracy. Use if `ode45` is slow (stiff systems) or there is a mass matrix.

- `ode23s`. Stiff problems, low accuracy. Use for large error tolerances with stiff systems or with a constant mass matrix.

- `ode23t`. Moderately stiff problems, low accuracy. Use for moderately stiff problems where you need a solution without numerical damping.

- `ode23tb`. Stiff problems, low accuracy. Use for large error tolerances with stiff systems or if there is a mass matrix.

7.13 Errors

Errors enter the numerical solution of the initial value problem from two sources:

- discretization error,

- roundoff error.

Discretization error is a property of the differential equation and the numerical method. If all the arithmetic could be performed with infinite precision, discretization error would be the only error present. Roundoff error is a property of the computer hardware and the program. It is usually far less important than the discretization error, except when we try to achieve very high accuracy.

Discretization error can be assessed from two points of view, local and global. *Local discretization error* is the error that would be made in one step if the previous values were exact and if there were no roundoff error. Let $u_n(t)$ be the solution of the differential equation determined not by the original initial condition at t_0 but

by the value of the computed solution at t_n. That is, $u_n(t)$ is the function of t defined by

$$\dot{u}_n = f(t, u_n),$$
$$u_n(t_n) = y_n.$$

The local discretization error d_n is the difference between this theoretical solution and the computed solution (ignoring roundoff) determined by the same data at t_n:

$$d_n = y_{n+1} - u_n(t_{n+1}).$$

Global discretization error is the difference between the computed solution, still ignoring roundoff, and the true solution determined by the original initial condition at t_0, that is,

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

The distinction between local and global discretization error can be easily seen in the special case where $f(t, y)$ does not depend on y. In this case, the solution is simply an integral, $y(t) = \int_{t_0}^{t} f(\tau)d\tau$. Euler's method becomes a scheme for numerical quadrature that might be called the "composite lazy man's rectangle rule." It uses function values at the left-hand ends of the subintervals rather than at the midpoints:

$$\int_{t_0}^{t_N} f(\tau)d\tau \approx \sum_{0}^{N-1} h_n f(t_n).$$

The local discretization error is the error in one subinterval:

$$d_n = h_n f(t_n) - \int_{t_n}^{t_{n+1}} f(\tau)d\tau,$$

and the global discretization error is the total error:

$$e_N = \sum_{n=0}^{N-1} h_n f(t_n) - \int_{t_0}^{t_N} f(\tau)d\tau.$$

In this special case, each of the subintegrals is independent of the others (the sum could be evaluated in any order), so the global error is the sum of the local errors:

$$e_N = \sum_{n=0}^{N-1} d_n.$$

In the case of a genuine differential equation where $f(t, y)$ depends on y, the error in any one interval depends on the solutions computed for earlier intervals. Consequently, the relationship between the global error and the local errors is related to the *stability* of the differential equation. For a single scalar equation, if the partial derivative $\partial f/\partial y$ is positive, then the solution $y(t)$ grows as t increases and the global error will be greater than the sum of the local errors. If $\partial f/\partial y$ is negative,

then the global error will be less than the sum of the local errors. If $\partial f/\partial y$ changes sign, or if we have a nonlinear system of equations where $\partial f/\partial y$ is a varying matrix, the relationship between e_N and the sum of the d_n can be quite complicated and unpredictable.

Think of the local discretization error as the deposits made to a bank account and the global error as the overall balance in the account. The partial derivative $\partial f/\partial y$ acts like an interest rate. If it is positive, the overall balance is greater than the sum of the deposits. If it is negative, the final error balance might well be less than the sum of the errors deposited at each step.

Our code `ode23tx`, like all the production codes in MATLAB, only attempts to control the local discretization error. Solvers that try to control estimates of the global discretization error are much more complicated, are expensive to run, and are not very successful.

A fundamental concept in assessing the accuracy of a numerical method is its *order*. The order is defined in terms of the local discretization error obtained if the method is applied to problems with smooth solutions. A method is said to be of order p if there is a number C so that

$$|d_n| \le C h_n^{p+1}.$$

The number C might depend on the partial derivatives of the function defining the differential equation and on the length of the interval over which the solution is sought, but it should be independent of the step number n and the step size h_n. The above inequality can be abbreviated using "big-oh notation":

$$d_n = O(h_n^{p+1}).$$

For example, consider Euler's method:

$$y_{n+1} = y_n + h_n f(t_n, y_n).$$

Assume the local solution $u_n(t)$ has a continuous second derivative. Then, using Taylor series near the point t_n,

$$u_n(t) = u_n(t_n) + (t - t_n)u_n'(t_n) + O((t - t_n)^2).$$

Using the differential equation and the initial condition defining $u_n(t)$,

$$u_n(t_{n+1}) = y_n + h_n f(t_n, y_n) + O(h_n^2).$$

Consequently,

$$d_n = y_{n+1} - u_n(t_{n+1}) = O(h_n^2).$$

We conclude that $p = 1$, so Euler's method is first order. The MATLAB naming conventions for ordinary differential equation solvers would imply that a function using Euler's method by itself, with fixed step size and no error estimate, should be called `ode1`.

Now consider the global discretization error at a fixed point $t = t_f$. As accuracy requirements are increased, the step sizes h_n will decrease, and the total number of steps N required to reach t_f will increase. Roughly, we shall have

$$N = \frac{t_f - t_0}{h},$$

where h is the average step size. Moreover, the global error e_N can be expressed as a sum of N local errors coupled by factors describing the stability of the equations. These factors do not depend in a strong way on the step sizes, and so we can say roughly that if the local error is $O(h^{p+1})$, then the global error will be $N \cdot O(h^{p+1}) = O(h^p)$. This is why $p+1$ was used instead of p as the exponent in the definition of order.

For Euler's method, $p=1$, so decreasing the average step size by a factor of 2 decreases the average local error by a factor of roughly $2^{p+1}=4$, but about twice as many steps are required to reach t_f, so the global error is decreased by a factor of only $2^p=2$. With higher order methods, the global error for smooth solutions is reduced by a much larger factor.

It should be pointed out that in discussing numerical methods for ordinary differential equations, the word "order" can have any of several different meanings. The order of a differential equation is the index of the highest derivative appearing. For example, $d^2y/dt^2 = -y$ is a second-order differential equation. The order of a system of equations sometimes refers to the number of equations in the system. For example, $\dot{y} = 2y - yz, \dot{z} = -z + yz$ is a second-order system. The order of a numerical method is what we have been discussing here. It is the power of the step size that appears in the expression for the global error.

One way of checking the order of a numerical method is to examine its behavior if $f(t, y)$ is a polynomial in t and does not depend on y. If the method is exact for t^{p-1}, but not for t^p, then its order is not more than p. (The order could be less than p if the method's behavior for general functions does not match its behavior for polynomials.) Euler's method is exact if $f(t, y)$ is constant, but not if $f(t, y) = t$, so its order is not greater than one.

With modern computers, using IEEE floating-point double-precision arithmetic, the roundoff error in the computed solution only begins to become important if very high accuracies are requested or the integration is carried out over a long interval. Suppose we integrate over an interval of length $L = t_f - t_0$. If the roundoff error in one step is of size ϵ, then the worst the roundoff error can be after N steps of size $h = \frac{L}{N}$ is something like

$$N\epsilon = \frac{L\epsilon}{h}.$$

For a method with global discretization error of size Ch^p, the total error is something like

$$Ch^p + \frac{L\epsilon}{h}.$$

For the roundoff error to be comparable with the discretization error, we need

$$h \approx \left(\frac{L\epsilon}{C}\right)^{\frac{1}{p+1}}.$$

The number of steps taken with this step size is roughly

$$N \approx L\left(\frac{C}{L\epsilon}\right)^{\frac{1}{p+1}}.$$

Here are the numbers of steps for various orders p if $L = 20$: $C = 100$, and $\epsilon = 2^{-52}$:

p	N
1	$4.5 \cdot 10^{17}$
3	5,647,721
5	37,285
10	864

These values of p are the orders for Euler's method and for the MATLAB functions `ode23` and `ode45`, and a typical choice for the order in the variable-order method used by `ode113`. We see that the low-order methods have to take an impractically large number of steps before this worst-case roundoff error estimate becomes significant. Even more steps are required if we assume the roundoff error at each step varies randomly. The variable-order multistep function `ode113` is capable of achieving such high accuracy that roundoff error can be a bit more significant with it.

7.14 Performance

We have carried out an experiment to see how all this applies in practice. The differential equation is the harmonic oscillator

$$\ddot{x}(t) = -x(t)$$

with initial conditions $x(0) = 1, \dot{x}(0) = 0$, over the interval $0 \le t \le 10\pi$. The interval is five periods of the periodic solution, so the global error can be computed simply as the difference between the initial and final values of the solution. Since the solution neither grows nor decays with t, the global error should be roughly proportional to the local error.

The following MATLAB script uses `odeset` to change both the relative and the absolute tolerances. The refinement level is set so that one step of the algorithm generates one row of output.

```
y0 = [1 0];
for k = 1:13
   tol = 10^(-k);
   opts = odeset('reltol',tol,'abstol',tol,'refine',1);
   tic
   [t,y] = ode23(@harmonic,[0 10*pi],y0',opts);
   time = toc;
   steps = length(t)-1;
   err = max(abs(y(end,:)-y0));
end
```

The differential equation is defined in `harmonic.m`.

```
function ydot = harmonic(t,y)
ydot = [y(2); -y(1)];
```

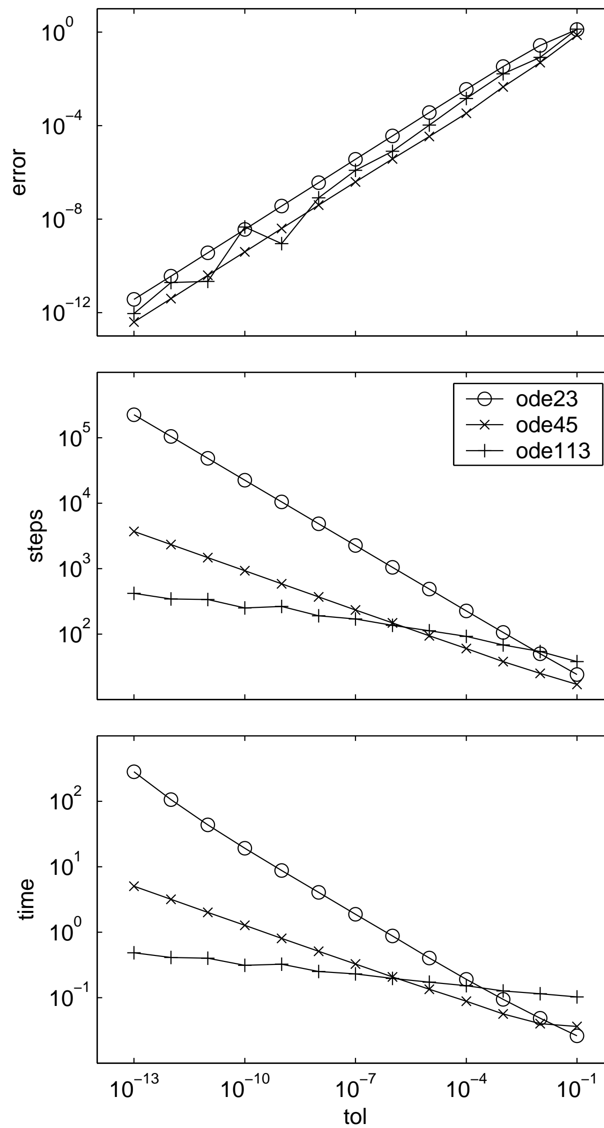

Figure 7.9. *Performance of ordinary differential equation solvers.*

The script was run three times, with `ode23`, `ode45`, and `ode113`. The first plot in Figure 7.9 shows how the global error varies with the requested tolerance for the three routines. We see that the actual error tracks the requested tolerance

quite well. For `ode23`, the global error is about 36 times the tolerance; for `ode45`, it is about 4 times the tolerance; and for `ode113`, it varies between 1 and 45 times the tolerance.

The second plot in Figure 7.9 shows the numbers of steps required. The results also fit our model quite well. Let τ denote the tolerance 10^{-k}. For `ode23`, the number of steps is about $10\tau^{-1/3}$, which is the expected behavior for a third-order method. For `ode45`, the number of steps is about $9\tau^{-1/5}$, which is the expected behavior for a fifth-order method. For `ode113`, the number of steps reflects the fact that the solution is very smooth, so the method was often able to use its maximum order, 13.

The third plot in Figure 7.9 shows the execution times, in seconds, on an 800 MHz Pentium III laptop. For this problem, `ode45` is the fastest method for tolerances of roughly 10^{-6} or larger, while `ode113` is the fastest method for more stringent tolerances. The low-order method, `ode23`, takes a very long time to obtain high accuracy.

This is just one experiment, on a problem with a very smooth and stable solution.

7.15 Further Reading

The MATLAB ordinary differential equation suite is described in [7]. Additional material on the numerical solution of ordinary differential equations, and especially stiffness, is available in Ascher and Petzold [1], Brennan, Campbell, and Petzold [2], and Shampine [6].

Exercises

7.1. The standard form of an ODE initial value problem is:

$$\dot{y} = f(t, y),\ \ y(t_0) = y_0.$$

Express this ODE problem in the standard form.

$$\ddot{u} = \frac{v}{1+t^2} - \sin r,$$
$$\ddot{v} = \frac{-u}{1+t^2} + \cos r,$$

where $r = \sqrt{\dot{u}^2 + \dot{v}^2}$. The initial conditions are

$$u(0) = 1, v(0) = \dot{u}(0) = \dot{v}(0) = 0.$$

7.2. You invest \$100 in a savings account paying 6% interest per year. Let $y(t)$ be the amount in your account after t years. If the interest is compounded continuously, then $y(t)$ solves the ODE initial value problem

$$\dot{y} = ry, \ r = .06$$

$$y(0) = 100.$$

Compounding interest at a discrete time interval, h, corresponds to using a finite difference method to approximate the solution to the differential equation. The time interval h is expressed as a fraction of a year. For example, compounding monthly has $h = 1/12$. The quantity y_n, the balance after n time intervals, approximates the continuously compounded balance $y(nh)$. The banking industry effectively uses Euler's method to compute compound interest.

$$y_0 = y(0),$$

$$y_{n+1} = y_n + hry_n.$$

This exercise asks you to investigate the use of higher order difference methods to compute compound interest. What is the balance in your account after 10 years with each of the following methods of compounding interest?

Euler's method, yearly.

Euler's method, monthly.

Midpoint rule, monthly.

Trapezoid rule, monthly.

BS23 algorithm, monthly.

Continuous compounding.

7.3. (a) Show experimentally or algebraically that the BS23 algorithm is exact for $f(t,y) = 1$, $f(t,y) = t$, and $f(t,y) = t^2$, but not for $f(t,y) = t^3$.
(b) When is the `ode23` error estimator exact?

7.4. The error function $\mathrm{erf}(x)$ is usually defined by an integral,

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-x^2} dx,$$

but it can also be defined as the solution to the differential equation

$$y'(x) = \frac{2}{\sqrt{\pi}} e^{-x^2},$$
$$y(0) = 0.$$

Use `ode23tx` to solve this differential equation on the interval $0 \le x \le 2$. Compare the results with the built-in MATLAB function `erf(x)` at the points chosen by `ode23tx`.

7.5. (a) Write an M-file named `myrk4.m`, in the style of `ode23tx.m`, that implements the classical Runge–Kutta fixed step size algorithm. Instead of an optional fourth argument `rtol` or `opts`, the required fourth argument should be the step size h. Here is the proposed preamble.

```
% function [tout,yout] = myrk4(F,tspan,y0,h,varargin)
% MYRK4  Classical fourth-order Runge-Kutta.
% Usage is the same as ODE23TX except the fourth
% argument is a fixed step size h.
% MYRK4(F,TSPAN,Y0,H) with TSPAN = [T0 TF] integrates
% the system of differential equations y' = f(t,y)
% from t = T0 to t = TF.  The initial condition
% is y(T0) = Y0.
% With no output arguments, MYRK4 plots the solution.
% With two output arguments, [T,Y] = MYRK4(..) returns
% T and Y so that Y(:,k) is the approximate solution at
% T(k). More than four input arguments,
% MYRK4(..,P1,P2,..), are passed on to F,
% F(T,Y,P1,P2,...).
```

(b) Roughly, how should the error behave if the step size h for classical Runge–Kutta is cut in half? (Hint: Why is there a "4" in the name of `myrk4`?) Run an experiment to illustrate this behavior.
(c) If you integrate the simple harmonic oscillator $\ddot{y} = -y$ over one full period, $0 \le t \le 2\pi$, you can compare the initial and final values of y to get a measure of the global accuracy. If you use your `myrk4` with a step size $h = \pi/50$, you should find that it takes 100 steps and computes a result with an error of about 10^{-6}. Compare this with the number of steps required by `ode23`, `ode45`, and `ode113` if the relative tolerance is set to 10^{-6} and the refinement level is set to one. This is a problem with a very smooth solution, so you should find that `ode23` requires more steps, while `ode45` and `ode113` require fewer.

7.6. The ordinary differential equation problem

$$\dot{y} = -1000(y - \sin t) + \cos t,\ \ y(0) = 1,$$

on the interval $0 \le t \le 1$ is mildly stiff.
(a) Find the exact solution, either by hand or using `dsolve` from the Symbolic Toolbox.
(b) Compute the solution with `ode23tx`. How many steps are required?
(c) Compute the solution with the stiff solver `ode23s`. How many steps are required?
(d) Plot the two computed solutions on the same graph, with line style `'.'` for the `ode23tx` solution and `'o'` for the `ode23s` solution.
(e) Zoom in, or change the axis settings, to show a portion of the graph where the solution is varying rapidly. You should see that both solvers are taking small steps.
(f) Show a portion of the graph where the solution is varying slowly. You should see that `ode23tx` is taking much smaller steps than `ode23s`.

7.7. The following problems all have the same solution on $0 \le t \le \pi/2$:

$$\dot{y} = \cos t,\ y(0) = 0,$$
$$\dot{y} = \sqrt{1 - y^2},\ y(0)0,$$
$$\ddot{y} = -y,\ y(0) = 0,\ \dot{y}(0) = 1,$$
$$\ddot{y} = -\sin t,\ y(0) = 0,\ \dot{y}(0) = 1.$$

(a) What is the common solution $y(t)$?
(b) Two of the problems involve second derivatives, $\ddot{y}$. Rewrite these problems as first-order systems, $\dot{y} = f(t, y)$, involving vectors y and f.
(c) What is the Jacobian, $J = \frac{\partial f}{\partial y}$, for each problem? What happens to each Jacobian as t approaches $\pi/2$?
(d) The work required by a Runge–Kutta method to solve an initial value problem $\dot{y} = f(t, y)$ depends on the function $f(t, y)$, not just the solution, $y(t)$. Use `odeset` to set both `reltol` and `abstol` to 10^{-9}. How much work does `ode45` require to solve each problem? Why are some problems more work than others?
(e) What happens to the computed solutions if the interval is changed to $0 \le t \le \pi$?
(f) What happens on $0 \le t \le \pi$ if the second problem is changed to

$$\dot{y} = \sqrt{|1 - y^2|},\ y(0) = 0.$$

7.8. Use the `jacobian` and `eig` functions in the Symbolic Toolbox to verify that the Jacobian for the two-body problem is

$$J = \frac{1}{r^5} \begin{bmatrix} 0 & 0 & r^5 & 0 \\ 0 & 0 & 0 & r^5 \\ 2y_1^2 - y_2^2 & 3y_1y_2 & 0 & 0 \\ 3y_1y_2 & 2y_2^2 - y_1^2 & 0 & 0 \end{bmatrix}$$

and that its eigenvalues are

$$\lambda = \frac{1}{r^{3/2}} \begin{bmatrix} \sqrt{2} \\ i \\ -\sqrt{2} \\ -i \end{bmatrix}.$$

7.9. Verify that the matrix in the Lorenz equations

$$A = \begin{bmatrix} -\beta & 0 & \eta \\ 0 & -\sigma & \sigma \\ -\eta & \rho & -1 \end{bmatrix}$$

is singular if and only if

$$\eta = \pm\sqrt{\beta(\rho - 1)}.$$

Verify that the corresponding null vector is

$$\begin{pmatrix} \rho - 1 \\ \eta \\ \eta \end{pmatrix}.$$

7.10. The Jacobian matrix J for the Lorenz equations is not A, but is closely related to A. Find J, compute its eigenvalues at one of the fixed points, and verify that the fixed point is unstable.

7.11. Find the largest value of ρ in the Lorenz equations for which the fixed point is stable.

7.12. All the values of ρ available with `lorenzgui` except $\rho = 28$ give trajectories that eventually settle down to stable periodic orbits. In his book on the Lorenz equations, Sparrow classifies a periodic orbit by what we might call its *signature*, a sequence of $+$'s and $-$'s specifying the order of the critical points that the trajectory circles during one period. A single $+$ or $-$ would be the signature of a trajectory that circles just one critical point, except that no such orbits exist. The signature '$+-$' indicates that the trajectory circles each critical point once. The signature '$+++-+---$' would indicate a very fancy orbit that circles the critical points a total of eight times before repeating itself.

What are the signatures of the four different periodic orbits generated by `lorenzgui`? Be careful—each of the signatures is different, and $\rho = 99.65$ is particularly delicate.

7.13. What are the periods of the periodic orbits generated for the different values of ρ available with `lorenzgui`?

7.14. The MATLAB `demos` directory contains an M-file, `orbitode`, that uses `ode45` to solve an instance of the *restricted three-body problem*. This involves the orbit of a light object around two heavier objects, such as an Apollo capsule around the earth and the moon. Run the demo and then locate its source code with the statements

```
orbitode
which orbitode
```

Make your own copy of `orbitode.m`. Find these two statements:

```
tspan = [0 7];
y0 = [1.2; 0; 0; -1.04935750983031990726];
```

These statements set the time interval for the integration and the initial position and velocity of the light object. Our question is, Where do these values come from? To answer this question, find the statement

```
[t,y,te,ye,ie] = ode45(@f,tspan,y0,options);
```

Remove the semicolon and insert three more statements after it:

```
te
ye
ie
```

Run the demo again. Explain how the values of `te`, `ye`, and `ie` are related to `tspan` and `y0`.

7.15. A classical model in mathematical ecology is the Lotka–Volterra predator-prey model. Consider a simple ecosystem consisting of rabbits that have an infinite supply of food and foxes that prey on the rabbits for their food. This is modeled by a pair of nonlinear, first-order differential equations:

$$\frac{dr}{dt} = 2r - \alpha rf, \; r(0) = r_0,$$
$$\frac{df}{dt} = -f + \alpha rf, \; f(0) = f_0,$$

where t is time, $r(t)$ is the number of rabbits, $f(t)$ is the number of foxes, and α is a positive constant. If $\alpha = 0$, the two populations do not interact, the rabbits do what rabbits do best, and the foxes die off from starvation. If $\alpha > 0$, the foxes encounter the rabbits with a probability that is proportional to the product of their numbers. Such an encounter results in a decrease in the number of rabbits and (for less obvious reasons) an increase in the number of foxes.

The solutions to this nonlinear system cannot be expressed in terms of other known functions; the equations must be solved numerically. It turns out that the solutions are always periodic, with a period that depends on the initial conditions. In other words, for any $r(0)$ and $f(0)$, there is a value $t = t_p$ when both populations return to their original values. Consequently, for all t,

$$r(t + t_p) = r(t), \; f(t + t_p) = f(t).$$

(a) Compute the solution with $r_0 = 300$, $f_0 = 150$, and $\alpha = 0.01$. You should find that t_p is close to 5. Make two plots, one of r and f as functions of t and one a phase plane plot with r as one axis and f as the other.

(b) Compute and plot the solution with $r_0 = 15$, $f_0 = 22$, and $\alpha = 0.01$. You should find that t_p is close to 6.62.

(c) Compute and plot the solution with $r_0 = 102$, $f_0 = 198$, and $\alpha = 0.01$. Determine the period t_p either by trial and error or with an event handler.

(d) The point $(r_0, f_0) = (1/\alpha, 2/\alpha)$ is a stable equilibrium point. If the populations have these initial values, they do not change. If the initial populations are close to these values, they do not change very much. Let $u(t) = r(t) - 1/\alpha$ and $v(t) = f(t) - 2/\alpha$. The functions $u(t)$ and $v(t)$ satisfy another nonlinear system of differential equations, but if the uv terms are ignored, the system becomes linear. What is this linear system? What is the period of its periodic solutions?

7.16. Many modifications of the Lotka–Volterra predator-prey model (see previous problem) have been proposed to more accurately reflect what happens in nature. For example, the number of rabbits can be prevented from growing indefinitely by changing the first equation as follows:

$$\frac{dr}{dt} = 2\left(1 - \frac{r}{R}\right) r - \alpha rf, \; r(0) = r_0,$$
$$\frac{df}{dt} = -f + \alpha rf, \; y(0) = y_0,$$

where t is time, $r(t)$ is the number of rabbits, $f(t)$ is the number of foxes, α is a positive constant, and R is a positive constant. Because α is positive, $\frac{dr}{dt}$ is negative whenever $r \geq R$. Consequently, the number of rabbits can never exceed R.

For $\alpha = 0.01$, compare the behavior of the original model with the behavior of this modified model with $R = 400$. In making this comparison, solve the equations with $r_0 = 300$ and $f_0 = 150$ over 50 units of time. Make four different plots:

- number of foxes and number of rabbits versus time for the original model,
- number of foxes and number of rabbits versus time for the modified model,
- number of foxes versus number of rabbits for the original model,
- number of foxes versus number of rabbits for the modified model.

For all plots, label all curves and all axes and put a title on the plot. For the last two plots, set the aspect ratio so that equal increments on the x- and y-axes are equal in size.

7.17. An 80-kg paratrooper is dropped from an airplane at a height of 600 m. After 5 s the chute opens. The paratrooper's height as a function of time, $y(t)$, is given by

$$\begin{aligned} \ddot{y} &= -g + \alpha(t)/m, \\ y(0) &= 600 \text{ m}, \\ \dot{y}(0) &= 0 \text{ m/s}, \end{aligned}$$

where $g = 9.81$ m/s^2 is the acceleration due to gravity and $m = 80$ kg is the paratrooper's mass. The air resistance $\alpha(t)$ is proportional to the square of the velocity, with different proportionality constants before and after the chute opens.

$$\alpha(t) = \begin{cases} K_1 \dot{y}(t)^2, & t < 5 \text{ s}, \\ K_2 \dot{y}(t)^2, & t \geq 5 \text{ s}. \end{cases}$$

(a) Find the analytical solution for free-fall, $K_1 = 0, K_2 = 0$. At what height does the chute open? How long does it take to reach the ground? What is the impact velocity? Plot the height versus time and label the plot appropriately.
(b) Consider the case $K_1 = 1/15, K_2 = 4/15$. At what height does the chute open? How long does it take to reach the ground? What is the impact velocity? Make a plot of the height versus time and label the plot appropriately.

7.18. Determine the trajectory of a spherical cannonball in a stationary Cartesian coordinate system that has a horizontal x-axis, a vertical y-axis, and an origin at the launch point. The initial velocity of the projectile in this coordinate system has magnitude v_0 and makes an angle with respect to the x-axis of θ_0 radians. The only forces acting on the projectile are gravity and the aerodynamic drag, D, which depends on the projectile's speed relative to any wind that might be present. The equations describing the motion of the

projectile are

$$\dot{x} = v \cos\theta, \quad \dot{y} = v \sin\theta,$$

$$\dot{\theta} = -\frac{g}{v}\cos\theta, \quad \dot{v} = -\frac{D}{m} - g\sin\theta.$$

Constants for this problem are the acceleration of gravity, $g = 9.81\text{m/s}^2$, the mass, $m = 15$ kg, and the initial speed, $v_0 = 50$ m/s. The wind is assumed to be horizontal and its speed is a specified function of time, $w(t)$. The aerodynamic drag is proportional to the square of the projectile's velocity relative to the wind:

$$D(t) = \frac{c\rho s}{2}\left((\dot{x} - w(t))^2 + \dot{y}^2\right),$$

where $c = 0.2$ is the drag coefficient, $\rho = 1.29$ kg/m^3 is the density of air, and $s = 0.25\,\text{m}^2$ is the projectile's cross-sectional area.
Consider four different wind conditions.

- No wind. $w(t) = 0$ for all t.
- Steady headwind. $w(t) = -10$ m/s for all t.
- Intermittent tailwind. $w(t) = 10$ m/s if the integer part of t is even, and zero otherwise.
- Gusty wind. $w(t)$ is a Gaussian random variable with mean zero and standard deviation 10 m/s.

The integer part of a real number t is denoted by $\lfloor t \rfloor$ and is computed in MATLAB by `floor(t)`. A Gaussian random variable with mean 0 and standard deviation σ is generated by `sigma*randn` (see Chapter 9, Random Numbers).
For each of these four wind conditions, carry out the following computations. Find the 17 trajectories whose initial angles are multiples of 5 degrees, that is, $\theta_0 = k\pi/36$ radians, $k = 1, 2, \ldots, 17$. Plot all 17 trajectories on one figure. Determine which of these trajectories has the greatest downrange distance. For that trajectory, report the initial angle in degrees, the flight time, the downrange distance, the impact velocity, and the number of steps required by the ordinary differential equation solver.
Which of the four wind conditions requires the most computation? Why?

7.19. In the 1968 Olympic games in Mexico City, Bob Beamon established a world record with a long jump of 8.90 m. This was 0.80 m longer than the previous world record. Since 1968, Beamon's jump has been exceeded only once in competition, by Mike Powell's jump of 8.95 m in Tokyo in 1991. After Beamon's remarkable jump, some people suggested that the lower air resistance at Mexico City's 2250 m altitude was a contributing factor. This problem examines that possibility.
The mathematical model is the same as the cannonball trajectory in the previous exercise. The fixed Cartesian coordinate system has a horizontal x-axis, a vertical y-axis, and an origin at the takeoff board. The jumper's initial velocity has magnitude v_0 and makes an angle with respect to the x-axis of θ_0 radians. The only forces acting after takeoff are gravity and the

aerodynamic drag, D, which is proportional to the square of the magnitude of the velocity. There is no wind. The equations describing the jumper's motion are

$$\dot{x} = v\cos\theta, \quad \dot{y} = v\sin\theta,$$

$$\dot{\theta} = -\frac{g}{v}\cos\theta, \quad \dot{v} = -\frac{D}{m} - g\sin\theta.$$

The drag is

$$D = \frac{c\rho s}{2}\left(\dot{x}^2 + \dot{y}^2\right).$$

Constants for this exercise are the acceleration of gravity, $g = 9.81$ m/s^2, the mass, $m = 80$ kg, the drag coefficient, $c = 0.72$, the jumper's cross-sectional area, $s = 0.50\,\mathrm{m}^2$, and the takeoff angle, $\theta_0 = 22.5^\circ = \pi/8$ radians.
Compute four different jumps, with different values for initial velocity, v_0, and air density, ρ. The length of each jump is $x(t_f)$, where the air time, t_f, is determined by the condition $y(t_f) = 0$.
(a) "Nominal" jump at high altitude. $v_0 = 10$ m/s and $\rho = 0.94$ kg/m^3.
(b) "Nominal" jump at sea level. $v_0 = 10$ m/s and $\rho = 1.29$ kg/m^3.
(c) Sprinter's approach at high altitude. $\rho = 0.94$ kg/m^3. Determine v_0 so that the length of the jump is Beamon's record, 8.90 m.
(d) Sprinter's approach at sea level. $\rho = 1.29$ kg/m^3 and v_0 is the value determined in (c).
Present your results by completing the following table.

```
   v0        theta0     rho       distance
 10.0000    22.5000    0.9400      ???
 10.0000    22.5000    1.2900      ???
   ???      22.5000    0.9400    8.9000
   ???      22.5000    1.2900      ???
```

Which is more important, the air density or the jumper's initial velocity?

7.20. A pendulum is a point mass at the end of a weightless rod of length L supported by a frictionless pin. If gravity is the only force acting on the pendulum, its oscillation is modeled by

$$\ddot{\theta} = -(g/L)\sin\theta.$$

Here θ is the angular position of the rod, with $\theta = 0$ if the rod is hanging down from the pin and $\theta = \pi$ if the rod is precariously balanced above the pin. Take $L = 30$ cm and $g = 981$ cm/s^2. The initial conditions are

$$\theta(0) = \theta_0,$$
$$\dot{\theta}(0) = 0.$$

If the initial angle θ_0 is not too large, then the approximation

$$\sin\theta \approx \theta$$

leads to a *linearized* equation

$$\ddot{\theta} = -(g/L)\theta$$

that is easily solved.
(a) What is the period of oscillation for the linearized equation?
If we do not make the assumption that θ_0 is small and do not replace $\sin\theta$ by θ, then it turns out that the period T of the oscillatory motion is given by

$$T(\theta_0) = 4(L/g)^{1/2} K(\sin^2(\theta_0/2)),$$

where $K(s^2)$ is the complete elliptic integral of the first kind, given by

$$K(s^2) = \int_0^1 \frac{dt}{\sqrt{1-s^2t^2}\,\sqrt{1-t^2}}.$$

(b) Compute and plot $T(\theta_0)$ for $0 \le \theta_0 \le 0.9999\pi$ two different ways. Use the MATLAB function `ellipke` and also use numerical quadrature with `quadtx`. Verify that the two methods yield the same results, to within the quadrature tolerance.
(c) Verify that for small θ_0 the linear equation and the nonlinear equation have approximately the same period.
(d) Compute the solutions to the nonlinear model over one period for several different values of θ_0, including values near 0 and near π. Superimpose the phase plane plots of the solutions on one graph.

7.21. What effect does the burning of fossil fuels have on the carbon dioxide in the earth's atmosphere? Even though today carbon dioxide accounts for only about 350 parts per million of the atmosphere, any increase has profound implications for our climate. An informative background article is available at a Web site maintained by the Lighthouse Foundation [5].
A model developed by J. C. G. Walker [9] was brought to our attention by Eric Roden. The model simulates the interaction of the various forms of carbon that are stored in three regimes: the atmosphere, the shallow ocean, and the deep ocean. The five principal variables in the model are all functions of time:

p, partial pressure of carbon dioxide in the atmosphere;

σ_s, total dissolved carbon concentration in the shallow ocean;

σ_d, total dissolved carbon concentration in the deep ocean;

α_s, alkalinity in the shallow ocean;

α_d, alkalinity in the deep ocean.

Three additional quantities are involved in equilibrium equations in the shallow ocean:

h_s, hydrogen carbonate in the shallow ocean;

c_s, carbonate in the shallow ocean;

p_s, partial pressure of gaseous carbon dioxide in the shallow ocean.

The rate of change of the five principal variables is given by five ordinary differential equations. The exchange between the atmosphere and the shallow ocean involves a constant characteristic transfer time d and a source term $f(t)$:

$$\frac{dp}{dt} = \frac{p_s - p}{d} + \frac{f(t)}{\mu_1}.$$

The equations describing the exchange between the shallow and deep oceans involve v_s and v_d, the volumes of the two regimes:

$$\begin{aligned}
\frac{d\sigma_s}{dt} &= \frac{1}{v_s}\left((\sigma_d - \sigma_s)w - k_1 - \frac{p_s - p}{d}\mu_2\right), \\
\frac{d\sigma_d}{dt} &= \frac{1}{v_d}\left(k_1 - (\sigma_d - \sigma_s)w\right), \\
\frac{d\alpha_s}{dt} &= \frac{1}{v_s}\left((\alpha_d - \alpha_s)w - k_2\right), \\
\frac{d\alpha_d}{dt} &= \frac{1}{v_d}\left(k_2 - (\alpha_d - \alpha_s)w\right).
\end{aligned}$$

The equilibrium between carbon dioxide and the carbonates dissolved in the shallow ocean is described by three nonlinear algebraic equations:

$$\begin{aligned}
h_s &= \frac{\sigma_s - \left(\sigma_s^2 - k_3\alpha_s(2\sigma_s - \alpha_s)\right)^{1/2}}{k_3}, \\
c_s &= \frac{\alpha_s - h_s}{2}, \\
p_s &= k_4\frac{h_s^2}{c_s}.
\end{aligned}$$

The numerical values of the constants involved in the model are

$$\begin{aligned}
d &= 8.64, \\
\mu_1 &= 4.95 \cdot 10^{2}, \\
\mu_2 &= 4.95 \cdot 10^{-2}, \\
v_s &= 0.12, \\
v_d &= 1.23, \\
w &= 10^{-3}, \\
k_1 &= 2.19 \cdot 10^{-4}, \\
k_2 &= 6.12 \cdot 10^{-5}, \\
k_3 &= 0.997148, \\
k_4 &= 6.79 \cdot 10^{-2}.
\end{aligned}$$

The source term $f(t)$ describes the burning of fossil fuels in the modern industrial era. We will use a time interval that starts about a thousand years ago and extends a few thousand years into the future:

$$1000 \le t \le 5000.$$

The initial values at $t = 1000$,

$$\begin{aligned} p &= 1.00, \\ \sigma_s &= 2.01, \\ \sigma_d &= 2.23, \\ \alpha_s &= 2.20, \\ \alpha_d &= 2.26, \end{aligned}$$

represent preindustrial equilibrium and remain nearly constant as long as the source term $f(t)$ is zero.

The following table describes one scenario for a source term $f(t)$ that models the release of carbon dioxide from burning fossil fuels, especially gasoline. The amounts begin to be significant after 1850, peak near the end of this century, and then decrease until the supply is exhausted.

year	rate
1000	0.0
1850	0.0
1950	1.0
1980	4.0
2000	5.0
2050	8.0
2080	10.0
2100	10.5
2120	10.0
2150	8.0
2225	3.5
2300	2.0
2500	0.0
5000	0.0

Figure 7.10 shows this source term and its effect on the atmosphere and the ocean. The three graphs in the lower half of the figure show the atmospheric, shallow ocean, and deep ocean carbon. (The two alkalinity values are not plotted at all because they are almost constant throughout this entire simulation.) Initially, the carbon in the three regimes is nearly at equilibrium and so the amounts hardly change before 1850.

Over the period $1850 \le t \le 2500$, the upper half of Figure 7.10 shows the additional carbon produced by burning fossil fuels entering the system, and the lower half shows the system response. The atmosphere is the first to be affected, showing more than a fourfold increase in 500 years. Almost half of the carbon is then slowly transferred to the shallow ocean and eventually to the deep ocean.

(a) Reproduce Figure 7.10. Use `pchiptx` to interpolate the fuel table and `ode23tx` with the default tolerances to solve the differential equations.
(b) How do the amounts of carbon in the three regimes at year 5000 compare with the amounts at year 1000?
(c) When does the atmospheric carbon dioxide reach its maximum?
(d) These equations are mildly stiff, because the various chemical reactions take place on very different time scales. If you zoom in on some portions of the graphs, you should see a characteristic sawtooth behavior caused by the small time steps required by `ode23tx`. Find such a region.
(e) Experiment with other MATLAB ordinary differential equation solvers, including `ode23`, `ode45`, `ode113`, `ode23s`, and `ode15s`. Try various tolerances and report computational costs by using something like

```
odeset('RelTol',1.e-6,'AbsTol',1.e-6,'stats','on');
```

Which method is preferable for this problem?

7.22. This problem makes use of quadrature, ordinary differential equations, and zero finding to study a nonlinear boundary value problem. The function $y(x)$ is defined on the interval $0 \leq x \leq 1$ by

$$y'' = y^2 - 1,$$

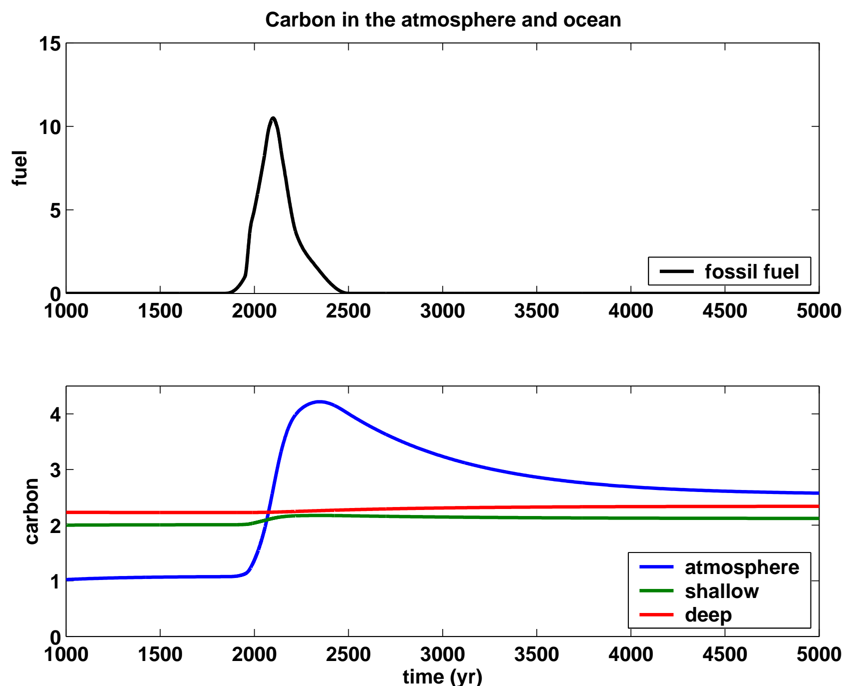

Figure 7.10. *Carbon in the atmosphere and ocean.*

$$y(0) = 0,$$
$$y(1) = 1.$$

This problem can be solved four different ways. Plot the four solutions obtained on a single figure, using `subplot(2,2,1),..., subplot(2,2,4)`.
(a) Shooting method. Suppose we know the value of $\eta = y'(0)$. Then we could use an ordinary differential equation solver like `ode23tx` or `ode45` to solve the initial value problem

$$y'' = y^2 - 1,$$
$$y(0) = 0,$$
$$y'(0) = \eta.$$

on the interval $0 \le x \le 1$. Each value of η determines a different solution $y(x;\eta)$ and corresponding value for $y(1;\eta)$. The desired boundary condition $y(1) = 1$ leads to the definition of a function of η:

$$f(\eta) = y(1;\eta) - 1.$$

Write a MATLAB function whose argument is η. This function should solve the ordinary differential equation initial problem and return $f(\eta)$. Then use `fzero` or `fzerotx` to find a value η_* so that $f(\eta_*) = 0$. Finally, use this η_* in the initial value problem to get the desired $y(x)$. Report the value of η_* you obtain.
(b) Quadrature. Observe that $y'' = y^2 - 1$ can be written

$$\frac{d}{dx}\left(\frac{(y')^2}{2} - \frac{y^3}{3} + y\right) = 0.$$

This means that the expression

$$\kappa = \frac{(y')^2}{2} - \frac{y^3}{3} + y$$

is actually constant. Because $y(0) = 0$, we have $y'(0) = \sqrt{2\kappa}$. So, if we could find the constant κ, the boundary value problem would be converted into an initial value problem. Integrating the equation

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

gives

$$x = \int_0^y h(y,\kappa)\, dy,$$

where

$$h(y,\kappa) = \frac{1}{\sqrt{2(\kappa + y^3/3 - y)}}.$$

This, together with the boundary condition $y(1) = 1$, leads to the definition of a function $g(\kappa)$:

$$g(\kappa) = \int_0^1 h(y, \kappa)\ dy\ -\ 1.$$

You need two MATLAB functions, one that computes $h(y, \kappa)$ and one that computes $g(\kappa)$. They can be two separate M-files, but a better idea is to make $h(y, \kappa)$ an inline function within $g(\kappa)$. The function $g(\kappa)$ should use `quadtx` to evaluate the integral of $h(y, \kappa)$. The parameter κ is passed as an extra argument from g, through `quadtx`, to h. Then `fzerotx` can be used to find a value κ_* so that $g(\kappa_*) = 0$. Finally, this κ_* provides the second initial value necessary for an ordinary differential equation solver to compute $y(x)$. Report the value of κ_* you obtain.

(c and d) Nonlinear finite differences. Partition the interval into $n+1$ equal subintervals with spacing $h = 1/(n+1)$:

$$x_i = ih,\ i = 0, \dots, n+1.$$

Replace the differential equation with a nonlinear system of difference equations involving n unknowns, $y_1, y_2, \dots, y_n$:

$$y_{i+1} - 2y_i + y_{i-1} = h^2(y_i^2 - 1),\ i = 1, \dots, n.$$

The boundary conditions are $y_0 = 0$ and $y_{n+1} = 1$.

A convenient way to compute the vector of second differences involves the n-by-n tridiagonal matrix A with -2's on the diagonal, 1's on the super- and subdiagonals, and 0's elsewhere. You can generate a sparse form of this matrix with

```
e = ones(n,1);
A = spdiags([e -2*e e],[-1 0 1],n,n);
```

The boundary conditions $y_0 = 0$ and $y_{n+1} = 1$ can be represented by the n-vector b, with $b_i = 0, i = 1, \dots, n-1$, and $b_n = 1$. The vector formulation of the nonlinear difference equation is

$$Ay + b = h^2(y^2 - 1),$$

where y^2 is the vector containing the squares of the elements of y, that is, the MATLAB element-by-element power `y.^2`. There are at least two ways to solve this system.

(c) Linear iteration. This is based on writing the difference equation in the form

$$Ay = h^2(y^2 - 1) - b.$$

Start with an initial guess for the solution vector y. The iteration consists of plugging the current y into the right-hand side of this equation and then solving the resulting linear system for a new y. This makes repeated use of the sparse backslash operator with the iterated assignment statement

```
y = A\(h^2*(y.^2 - 1) - b)
```

It turns out that this iteration converges linearly and provides a robust method for solving the nonlinear difference equations. Report the value of n you use and the number of iterations required.

(d) Newton's method. This is based on writing the difference equation in the form

$$F(y) = Ay + b - h^2(y^2 - 1) = 0.$$

Newton's method for solving $F(y) = 0$ requires a many-variable analogue of the derivative $F'(y)$. The analogue is the Jacobian, the matrix of partial derivatives

$$J = \frac{\partial F_i}{\partial y_j} = A - h^2 \mathrm{diag}(2y).$$

In MATLAB, one step of Newton's method would be

```
F = A*y + b - h^2*(y.^2 - 1);
J = A - h^2*spdiags(2*y,0,n,n);
y = y - J\F;
```

With a good starting guess, Newton's method converges in a handful of iterations. Report the value of n you use and the number of iterations required.

7.23. The double pendulum is a classical physics model system that exhibits chaotic motion if the initial angles are large enough. The model, shown in Figure 7.11, involves two weights, or *bobs*, attached by weightless, rigid rods to each other and to a fixed pivot. There is no friction, so once initiated, the motion continues forever. The motion is fully described by the two angles θ_1 and θ_2 that the rods make with the negative y-axis.

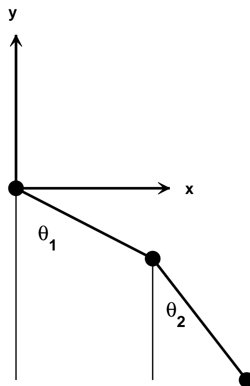

Figure 7.11. *Double pendulum.*

Let m_1 and m_2 be the masses of the bobs and ℓ_1 and ℓ_2 be the lengths of the

rods. The positions of the bobs are

$$x_1 = \ell_1 \sin\theta_1, \quad y_1 = -\ell_1 \cos\theta_1,$$
$$x_2 = \ell_1 \sin\theta_1 + \ell_2 \sin\theta_2, \quad y_2 = -\ell_1 \cos\theta_1 - \ell_2 \cos\theta_2.$$

The only external force is gravity, denoted by g. Analysis based on the Lagrangian formulation of classical mechanics leads to a pair of coupled, second-order, nonlinear ordinary differential equations for the two angles $\theta_1(t)$ and $\theta_2(t)$:

$$\begin{aligned}(m_1 + m_2)\ell_1\ddot{\theta}_1 + m_2\ell_2\ddot{\theta}_2 \cos(\theta_1 - \theta_2) &= -g(m_1 + m_2)\sin\theta_1 \\ &\quad -m_2\ell_2\dot{\theta}_2^2 \sin(\theta_1 - \theta_2), \\ m_2\ell_1\ddot{\theta}_1 \cos(\theta_1 - \theta_2) + m_2\ell_2\ddot{\theta}_2 &= -gm_2 \sin\theta_2 + m_2\ell_1\dot{\theta}_1^2 \sin(\theta_1 - \theta_2).\end{aligned}$$

To rewrite these equations as a first-order system, introduce the 4-by-1 column vector $u(t)$:

$$u = [\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2]^T.$$

With $m_1 = m_2 = \ell_1 = \ell_2 = 1$, $c = \cos(u_1 - u_2)$, and $s = \sin(u_1 - u_2)$, the equations become

$$\begin{aligned}\dot{u}_1 &= u_3, \\ \dot{u}_2 &= u_4, \\ 2\dot{u}_3 + c\dot{u}_4 &= -g\sin u_1 - su_4^2, \\ c\dot{u}_3 + \dot{u}_4 &= -g\sin u_2 + su_3^2.\end{aligned}$$

Let $M = M(u)$ denote the 4-by-4 *mass matrix*

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & c \\ 0 & 0 & c & 1 \end{pmatrix}$$

and let $f = f(u)$ denote the 4-by-1 nonlinear *force function*

$$f = \begin{pmatrix} u_3 \\ u_4 \\ -g\sin u_1 - su_4^2 \\ -g\sin u_2 + su_3^2 \end{pmatrix}.$$

In matrix-vector notation, the equations are simply

$$M\dot{u} = f.$$

This is an *implicit* system of differential equations involving a nonconstant, nonlinear mass matrix. The double pendulum problem is usually formulated without the mass matrix, but larger problems, with more degrees of freedom,

are frequently in implicit form. In some situations, the mass matrix is singular and it is not possible to write the equations in explicit form.
The NCM M-file `swinger` provides an interactive graphical implementation of these equations. The initial position is determined by specifying the starting coordinates of the second bob, (x_2, y_2), either as arguments to `swinger` or by using the mouse. In most situations, this does not uniquely determine the starting position of the first bob, but there are only two possibilities and one of them is chosen arbitrarily. The initial velocities, $\dot{\theta}_1$ and $\dot{\theta}_2$, are zero.
The numerical solution is carried out by `ode23` because our textbook code, `ode23tx`, cannot handle implicit equations. The call to `ode23` involves using `odeset` to specify the functions that generate the mass matrix and do the plotting

```
opts = odeset('mass',@swingmass, ...
       'outputfcn',@swingplot);
ode23(@swingrhs,tspan,u0,opts);
```

The mass matrix function is

```
function M = swingmass(t,u)
c = cos(u(1)-u(2));
M = [1 0 0 0; 0 1 0 0; 0 0 2 c; 0 0 c 1];
```

The driving force function is

```
function f = swingrhs(t,u)
g = 1;
s = sin(u(1)-u(2));
f = [u(3); u(4); -2*g*sin(u(1))-s*u(4)^2;
     -g*sin(u(2))+s*u(3)^2];
```

It would be possible to have just one ordinary differential equation function that returns `M\f`, but we want to emphasize the implicit facility.
An internal function `swinginit` converts a specified starting point (x, y) to a pair of angles (θ_1, θ_2). If (x, y) is outside the circle

$$\sqrt{x^2 + y^2} > \ell_1 + \ell_2,$$

then the pendulum cannot reach the specified point. In this case, we straighten out the pendulum with $\theta_1 = \theta_2$ and point it in the given direction. If (x, y) is inside the circle of radius two, we return one of the two possible configurations that reach to that point.
Here are some questions to guide your investigation of `swinger`.
(a) When the initial point is outside the circle of radius two, the two rods start out as one. If the initial angle is not too large, the double pendulum continues to act pretty much like a single pendulum. But if the initial angles are large enough, chaotic motion ensues. Roughly what initial angles lead to chaotic motion?
(b) The default initial condition is

```
swinger(0.862,-0.994)
```

Why is this orbit interesting? Can you find any similar orbits?
(c) Run `swinger` for a while, then click on its `stop` button. Go to the MATLAB command line and type `get(gcf,'userdata')`. What is returned?
(d) Modify `swinginit` so that, when the initial point is inside the circle of radius two, the other possible initial configuration is chosen.
(e) Modify `swinger` so that masses other than $m_1 = m_2 = 1$ are possible.
(f) Modify `swinger` so that lengths other than $\ell_1 = \ell_2 = 1$ are possible. This is trickier than changing the masses because the initial geometry is involved.
(g) What role does gravity play? How would the behavior of a double pendulum change if you could take it to the moon? How does changing the value of `g` in `swingrhs` affect the speed of the graphics display, the step sizes chosen by the ordinary differential equation solver, and the computed values of `t`?
(h) Combine `swingmass` and `swingrhs` into one function, `swingode`. Eliminate the `mass` option and use `ode23tx` instead of `ode23`.
(i) Are these equations stiff?
(j) This is a difficult question. The statement `swinger(0,2)` tries to delicately balance the pendulum above its pivot point. The pendulum does stay there for a while, but then loses its balance. Observe the value of t displayed in the title for `swinger(0,2)`. What force knocks the pendulum away from the vertical position? At what value of t does this force become noticeable?

Bibliography

[1] U. M. Ascher and L. R. Petzold, *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*, SIAM, Philadelphia, 1998.

[2] K. E. Brenan, S. L. Campbell, and L. R. Petzold, *Numerical Solution of Initial Value Problems in Differential-Algebraic Equations*, SIAM, Philadelphia, 1996.

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

[4] R. M. Corless, G. H. Gonnet, D. E. G. Hare, D. J. Jeffrey, and D. E. Knuth, *On the Lambert W function*, Advances in Computational Mathematics, 5 (1996), pp. 329–359.
`http://www.apmaths.uwo.ca/~rcorless/frames/PAPERS/LambertW`

[5] Lighthouse Foundation.
`http://www.lighthouse-foundation.org/lighthouse-foundation.org/eng/explorer/artikel00294eng.html`

[6] L. F. Shampine, *Numerical Solution of Ordinary Differential Equations*, Chapman and Hall, New York, 1994.

[7] L. F. Shampine and M. W. Reichelt, *The MATLAB ODE suite*, SIAM Journal on Scientific Computing, 18 (1997), pp. 1–22.

[8] C. Sparrow, *The Lorenz Equations: Bifurcations, Chaos, and Strange Attractors*, Springer-Verlag, New York, 1982.

[9] J. C. G. Walker, *Numerical Adventures with Geochemical Cycles*, Oxford University Press, New York, 1991.