Table 12.1. *MATLAB's ODE solvers.*

Solver	Problem type	Type of algorithm
ode45	Nonstiff	Explicit Runge–Kutta pair, orders 4 and 5
ode23	Nonstiff	Explicit Runge–Kutta pair, orders 2 and 3
ode113	Nonstiff	Explicit linear multistep, orders 1 to 13
ode15s	Stiff	Implicit linear multistep, orders 1 to 5
ode23s	Stiff	Modified Rosenbrock pair (one-step), orders 2 and 3
ode23t	Mildly stiff	Trapezoidal rule (implicit), orders 2 and 3
ode23tb	Stiff	Implicit Runge–Kutta type algorithm, orders 2 and 3

return four solution values at equally spaced points over each "natural" step. The default interpolation level can be overridden via the `Refine` property with `odeset`.

A full list of MATLAB's ODE solvers is given in Table 12.1. The authors of these solvers, Shampine and Reichelt, discuss some of the theoretical and practical issues that arose during their development in [72]. The functions are designed to be interchangeable in basic use. So, for example, the illustrations in the previous subsection continue to work if `ode45` is replaced by any of the other solvers. The functions mainly differ in (a) their efficiency on different problem types and (b) their capacity for accepting information about the problem in connection with Jacobians and mass matrices. With regard to efficiency, Shampine and Reichelt write in [72]:

> The experiments reported here and others we have made suggest that except in special circumstances, `ode45` should be the code tried first. If there is reason to believe the problem to be stiff, or if the problem turns out to be unexpectedly difficult for `ode45`, the `ode15s` code should be tried.

The stiff solvers in Table 12.1 use information about the Jacobian matrix, $\partial f_i/\partial y_j$, at various points along the solution. By default, they automatically generate approximate Jacobians using finite differences. However, the reliability and efficiency of the solvers is generally improved if a function that evaluates the Jacobian is supplied. Further options are also available for providing information about whether the Jacobian is sparse, constant or written in vectorized form. To illustrate how Jacobian information can be encoded, we look at the system of ODEs

$$\frac{d}{dt}y(t) = Ay(t) + y(t).*(1 - y(t)) + v,$$

where A is N-by-N and v is N-by-1 with

$$A = r_1 \begin{bmatrix} 0 & 1 & & & \\ -1 & 0 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & 1 \\ & & & -1 & 0 \end{bmatrix} + r_2 \begin{bmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & 1 \\ & & & 1 & -2 \end{bmatrix},$$

$v = [r_2 - r_1, 0, \ldots, 0, r_2 + r_1]^T$, $r_1 = -a/(2\Delta x)$ and $r_2 = b/\Delta x^2$. Here, a, b and Δx are parameters with values $a = 1$, $b = 5 \times 10^{-2}$ and $\Delta x = 1/(N+1)$. This ODE system

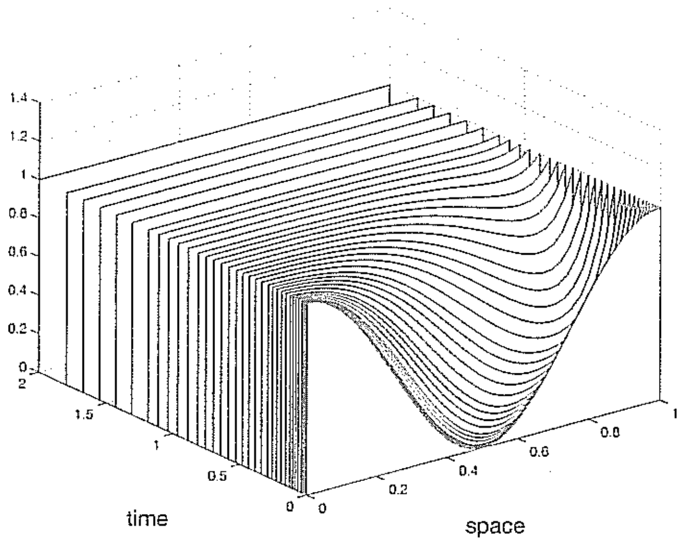

Figure 12.10. *Stiff ODE example, with Jacobian information supplied.*

arises when the method of lines based on central differences is used to semi-discretize the partial differential equation (PDE)

$$\frac{\partial}{\partial t}u(x,t) + a\frac{\partial}{\partial x}u(x,t) = b\frac{\partial^2}{\partial x^2}u(x,t) + u(x,t)(1-u(x,t)), \quad 0 \le x \le 1,$$

with Dirichlet boundary conditions $u(0,t) = u(1,t) = 1$. This PDE is of reaction-convection-diffusion type (and could be solved directly with pdepe, described in Section 12.4). The ODE solution component $y_j(t)$ approximates $u(j\Delta x, t)$. We suppose that the PDE comes with the initial data $u(x,0) = (1+\cos 2\pi x)/2$, for which it can be shown that $u(x,t)$ tends to the steady state $u(x,t) \equiv 1$ as $t \to \infty$. The corresponding ODE initial condition is $(y_0)_j = (1+\cos(2\pi j/(N+1)))/2$. The Jacobian for this ODE has the form $A + I - 2\,\mathrm{diag}(y(t))$, where I denotes the identity.

Listing 12.2 shows a function `rcd` that implements and solves this system using `ode15s`. It illustrates how a complete problem specification and solution can be encapsulated in a single function, by making use of subfunctions and function handles. We have set $N = 38$ and $0 \le t \le 2$. We specify via the `Jacobian` property of `odeset` the subfunction `jacobian` that evaluates the Jacobian, and the sparsity pattern of the Jacobian, encoded as a sparse matrix of 0s and 1s, is assigned to the `Jpattern` property. See Chapter 15 for details about sparse matrices and the function `spdiags`. The jth column of the output matrix `y` contains the approximation to $y_j(t)$, and we have created `U` by appending an extra column `ones(size(t))` at each end of `y` to account for the PDE boundary conditions. The plot produced by `rcd` is shown in Figure 12.10.

The ODE solvers can be applied to problems of the form

$$M(t,y(t))\frac{d}{dt}y(t) = f(t,y(t)), \quad y(t_0) = y_0,$$

where the *mass matrix*, $M(t,y(t))$, is square and nonsingular. (The `ode23s` solver applies only when M is independent of t and $y(t)$.) Mass matrices arise naturally when

Listing 12.2. *Function* rcd.

```
function rcd
%RCD Stiff ODE from method of lines on reaction-convection-diffusion problem.

N = 38; a = 1; b = 5e-2;
tspan = [0;2]; space = [1:N]/(N+1);

y0 = 0.5*(1+cos(2*pi*space));
y0 = y0(:);
options = odeset('Jacobian',@jacobian,'Jpattern',jpattern(N));
options = odeset(options,'RelTol',1e-3,'AbsTol',1e-3);

[t,y] = ode15s(@f,tspan,y0,options,N,a,b);
e = ones(size(t)); U = [e y e];
waterfall([0:1/(N+1):1],t,U)
xlabel('space','FontSize',16), ylabel('time','FontSize',16)

% ---------------------------------------------------------------------
% Subfunctions.
% ---------------------------------------------------------------------
function dydt = f(t,y,N,a,b)
%F         Differential equation.

r1 = -a*(N+1)/2;
r2 = b*(N+1)^2;
up = [y(2:N);0]; down = [0;y(1:N-1)];
e1 = [1;zeros(N-1,1)]; eN = [zeros(N-1,1);1];

dydt = r1*(up-down) + r2*(-2*y+up+down) + (r2-r1)*e1 + (r2+r1)*eN + y.*(1-y);

% ---------------------------------------------------------------------
function dfdy = jacobian(t,y,N,a,b)
%JACOBIAN  Jacobian matrix.

r1 = -a*(N+1)/2;
r2 = b*(N+1)^2;
u = (r2-r1)*ones(N,1);
v = (-2*r2+1)*ones(N,1) - 2*y;
w = (r2+r1)*ones(N,1);

dfdy = spdiags([u v w],[-1 0 1],N,N);

% ---------------------------------------------------------------------
function S = jpattern(N)
%JPATTERN  Sparsity pattern of Jacobian matrix.

e = ones(N,1);
S = spdiags([e e e],[-1 0 1],N,N);
```

semi-discretization is performed with a finite element method. A mass matrix can be specified in a similar manner to a Jacobian, via `odeset`. The `ode15s` and `ode23t` functions can solve certain problems where M is singular but does not depend on $y(t)$—more precisely, they can be used if the resulting differential-algebraic equation is of index 1 and y_0 is close to being consistent.

The ODE solvers offer other features that you may find useful. Type `help odeset` to see the full range of properties that can be controlled through the `options` structure. The function `odeget` extracts the current value of the `options` structure. The MATLAB ODE solvers are well documented and are supported by a rich variety of example files, some of which we list below. In each case, `help filename` gives an informative description of the file, `type filename` lists the contents of the file, and typing `filename` runs a demonstration.

`rigidode`: nonstiff ODE.

`brussode`, `vdpode`: stiff ODEs.

`ballode`: event location problem.

`orbitode`: problem involving event location and the use of an output function (`odephas2`) to process the solution as the integration proceeds.

`fem1ode`, `fem2ode`, `batonode`: ODEs with mass matrices.

`hb1dae`, `amp1dae`: differential-algebraic equations.

Type `odedemo` to run the example ODEs from a Graphical User Interface that offers a choice of solvers and plots the solutions.

12.3. Boundary Value Problems with bvp4c

The function `bvp4c` uses a collocation method to solve systems of ODEs in two-point boundary value form. These systems may be written

$$\frac{d}{dx}y(x) = f(x, y(x)), \quad g(y(a), y(b)) = 0.$$

Here, as for the initial value problem in the previous section, $y(x)$ is an unknown m-vector and f is a given function of x and y that also produces an m-vector. The solution is required over the range $a \le x \le b$ and the given function g specifies the boundary conditions. Note that the independent variable was labeled t in the previous section and is now labeled x. This is consistent with MATLAB's documentation and reflects the fact that two-point boundary value problems (BVPs) usually arise over an interval of space rather than time. Generally, BVPs are more computationally challenging than initial value problems. In particular, it is common for more than one solution to exist. For this reason, `bvp4c` requires an initial guess to be supplied for the solution. The initial guess and the final solution are stored in structures (see Section 18.3). We introduce `bvp4c` through a simple example before giving more details.

A scalar BVP describing the cross-sectional shape of a water droplet on a flat surface is given by [66]

$$\frac{d^2}{dx^2}h(x) + (1 - h(x))\left(1 + \left(\frac{d}{dx}h(x)\right)^2\right)^{3/2} = 0, \quad h(-1) = 0, \; h(1) = 0.$$