Chapter 2
More on ODE Integration

In Chap. 1, we considered through examples several numerical ODE integrators, e.g., Euler, leap-frog, Heun, Runge Kutta, and BDF methods. As we might expect, the development and use of ODE integrators is an extensive subject, and in this introductory text, we will only try to gain some overall perspective. To this end, we will first focus our attention to the `MATLAB` coding of a few basic integrators, i.e., a fixed-step Euler's method, a variable-step (nonstiff) Runge–Kutta (RK) method and a variable-step (stiff) Rosenbrock's method, and then turn our attention to more sophisticated library integrators. Typically, library integrators are written by experts who have included features that make the integrators robust and reliable. However, the coding of library integrators is generally long and relatively complicated, so that they are often used without a detailed understanding of how they work. A popular library is the `MATLAB ODE SUITE` developed by Shampine et al. [1]. Other options are provided by the use of `MATLAB` MEX-files and of readily available integrators originally written in `FORTRAN` or `C/C++` language. On the other hand, SCILAB provides a library of solvers based on ODEPACK [2] whereas OCTAVE includes LSODE [3] and DASSL [4]. We will explore these several options and discuss a few more ODE applications.

2.1 A Basic Fixed Step ODE Integrator

We considered previously the Euler's method—see Eqs. (1.15) and (1.16). A function that implements Euler's method with a fixed or constant integration step is listed in function `Euler_solver`.

A. Vande Wouwer et al., *Simulation of ODE/PDE Models with MATLAB®, OCTAVE and SCILAB*, DOI: 10.1007/978-3-319-06790-2_2,
© Springer International Publishing Switzerland 2014

```
function [tout,xout] = euler_solver(odefunction, t0, tf, x0,...
                                    Dt, Dtplot)

% This function solves first-order differential equations using
% Euler's method.
% [tout,xout] = euler_solver(@f,t0,tf,x0,Dt,Dtplot)
% integrates the system of differential equations xt=f(t,x) from
% t0 to tf with initial conditions x0.  f is a string containing
% the name of an ODE file.  Function f(t,x) must return a column
% vector.  Each row in solution array x corresponds to a value
% returned in column vector t.
%
% Argument list
%
% f - string containing the name of the user-supplied problem
%     call: xt = problem_name(t,x) where f = 'problem_name'
%     t - independent variable (scalar)
%     x - solution vector
%     xt - returned derivative vector; xt(i) = dx(i)/dt
% t0 - initial value of t
% tf - final value of t
% x0 - initial value vector
% Dt - time step size
% Dtplot - plot interval
% tout - returned integration points (column-vector).
% xout - returned solution, one solution row-vector per tout-value

% Initialization
plotgap = round(Dtplot/Dt);      % number of computation

% steps within a plot interval
Dt     = Dtplot/plotgap;
nplots = round((tf-t0)/Dtplot);  % number of plots
t      = t0;                     % initialize t
x      = x0;                     % initialize x
tout   = t0;                     % initialize output value
xout   = x0';                    % initialize output value

% Implement Euler's method
for i = 1: nplots
    for j = 1: plotgap
        % Use MATLAB's feval function to access the function
        % file, then take Euler step
        xnew = x + feval(odefunction,t,x)*Dt;
        t    = t + Dt;
        x    = xnew;
    end
    % Add latest result to the output arrays
    tout = [tout;t];
    xout = [xout;x'];
end
```

Function Euler_solver Basic Euler integrator

We can note the following points about this code:

1. The operation of function `Euler_solver`, and its input and output arguments are generally explained and defined in a block of comments.
2. The number of computation steps within a plot (output) interval, `plotgap`, the fixed-length Euler step, `Dt`, and the number of plot points (outputs), `nplots`, are first computed to take the integration from the initial value `t0` to the final value `tf`.
3. The initial conditions corresponding to Eq. (1.13) are then stored. Note that `t`, `t0` and `tout` are scalars while `x`, `x0` and `xout` are one-dimensional vectors.
4. `nplots` plot (output) steps are then taken using an outer `for` loop, and `plotgap` Euler steps are taken within each plot step using an inner `for` loop.
5. The solution is then advanced by Euler's method (1.16). Note that a `MATLAB` function can be evaluated using the utility `feval`. In the present case, the RHS of (1.12) is coded in function `odefunction`. The solution is then updated for the next Euler step. Finally, this code completes the inner for `loop` to give the solution at a plot point after `plotgap` Euler steps.
6. The solution is then stored in the arrays `tout` and `xout`. Thus, `tout` becomes a column vector and `xout` is a two-dimensional array with a column dimension the same as `tout` and each row containing the dependent variable vector for the corresponding `tout` (note that the latest solution is transposed into a row of `xout`).

The basic Euler integrator (see function `Euler_solver`) can now be called in any new application, with the particular ODEs defined in a function. To illustrate this approach, we consider the *logistic equation* which models population growth (N represents the number of individuals):

$$\frac{\mathrm{d}N}{\mathrm{d}t} = (a - bN)N = aN - bN^2 \tag{2.1}$$

first proposed by Verhulst in 1838 [5]. Equation (2.1) is a generalization of the ODE

$$\frac{\mathrm{d}N}{\mathrm{d}t} = aN \tag{2.2}$$

which for the Malthusian rate $a > 0$ gives unlimited exponential growth, i.e., the solution to Eq. (2.2) is

$$N(t) = N_0 e^{at} \tag{2.3}$$

where N_0 is an initial value of N for $t = 0$.

Since, realistically, unlimited growth is not possible in any physical process, we now consider Eq. (2.1) as an extension of Eq. (2.2) to reflect limited growth. Thus, instead of $\frac{\mathrm{d}N}{\mathrm{d}t} \to \infty$ as $N \to \infty$ according to Eq. (2.2), the solution now reaches the *equilibrium condition*

$$\frac{\mathrm{d}N}{\mathrm{d}t} = 0 = (a - bN)N$$

corresponding to $N = \frac{a}{b}$.

The approach to this equilibrium can be understood by considering the RHS of Eq. (2.1). At the beginning of the solution, for small t and N, the term aN dominates, and the solution grows according to Eqs. (2.2) and (2.3). For large t, N increases so that $-bN^2$ begins to offset aN and eventually the two terms become equal (but opposite in sign), at which point $\frac{\mathrm{d}N}{\mathrm{d}t} = 0$. Thus, the solution to Eq. (2.1) shows rapid growth initially, then slower growth, to produce a S-shaped solution.

These features are demonstrated by the analytical solution to Eq. (2.1), which can be derived as follows. First, *separation of variables* can be applied:

$$\frac{\mathrm{d}N}{(a - bN)N} = \mathrm{d}t \tag{2.4}$$

The LHS of Eq. (2.4) can then be expanded into two terms by *partial fractions*

$$\frac{\mathrm{d}N}{(a - bN)N} = \frac{\frac{\mathrm{d}N}{a}}{N} + \frac{\frac{b\mathrm{d}N}{a}}{a - bN} = \mathrm{d}t \tag{2.5}$$

Integration of Eq. (2.5) gives

$$\frac{1}{a}\ln(N) - \frac{b}{a}\ln(|a - bN|) = t + c \tag{2.6}$$

where c is a constant of integration that can be evaluated from the initial condition

$$N(0) = N_0 \tag{2.7}$$

so that Eq. (2.6) can be rewritten as

$$\frac{1}{a}\ln(N) - \frac{1}{a}\ln(|a - bN|) = t + \frac{1}{a}\ln(N_0) - \frac{b}{a}\ln(|a - bN_0|)$$

or, since $a - bN_0$ and $a - bN$ have the same sign,

$$\ln\left(\frac{N}{N_0}\right) + \ln\left(\frac{a - bN_0}{a - bN}\right) = at$$

and thus the analytical solution of Eq. (2.1) is

$$\left(\frac{N}{N_0}\right)\left(\frac{a - bN_0}{a - bN}\right) = e^{at}$$

or solving for N

$$N = \frac{\frac{a}{b}}{1 + \left(\frac{a - bN_0}{bN_0}\right) e^{-at}} = \frac{K}{1 + \left(\frac{K}{N_0} - 1\right) e^{-at}}, \quad K = \frac{a}{b} \tag{2.8}$$

Note again from Eq. (2.8) that for $a > 0, b > 0$, we have that as $t \to \infty, N \to \frac{a}{b} = K$, where K is called the *carrying capacity*.

Equation (2.8) can be used to evaluate the numerical solution from function `Euler_solver`. To do this, a function must be provided to define the ODE.

```
function Nt = logistic_ode(t,N)

% Global variables
global a b K N0

% ODE
Nt = (a-b*N)*N;
```

Function logistic_ode Function to define the ODE of the logistic equation (2.1)

The coding of function `logistic_ode` follows directly from Eq. (2.1). Note that this ODE function must return the time derivative of the state variable (`Nt` in this case). The ODE function is called by `Euler_solver`, which is called by the main program `Main_logistic`.

```
clear all
close all

% Set global variables
global a b K N0

% Model parameters
a = 1;
b = 0.5e-4;
K = a/b;

% Initial conditions
t0     = 0;
N0     = 1000;
tf     = 15;
Dt     = 0.1;
Dtplot = 0.5;

% Call to ODE solver
[tout,xout] = euler_solver(@logistic_ode,t0,tf,N0,Dt,Dtplot);

% Plot results
figure(1)
```

```
plot(tout,xout,'k');
hold on
Nexact = logistic_exact(tout);
plot(tout,Nexact,':r')
xlabel('t');
xlabel('N(t)');
title('Logistic equation');
```

Script Main_logistic Main program that calls functions `Euler_solver` and `logistic_ode`

This program proceeds in several steps:

1. After clearing `MATLAB` for a new application (via `clear` and `close`), the initial and final value of the independent variable, `t`, and the initial condition for N (`N0`) are defined.
2. The fixed integration step and plot (output) intervals are then defined. Thus, function `Euler_solver` will take $(15-0)/0.1 = 150$ Euler steps and the solution will be plotted $(15-0)/0.5+1 = 31$ times (counting the initial condition). Within each `Dtplot` step (outer `for` loop), `Euler_solver` will take $0.5/0.1 = 5$ Euler steps (inner `for` loop).
3. Function `Euler_solver` is then called with the preceding parameters and the solution returned as the one-dimensional column vector `tout` and the matrix `xout` (one-dimensional with the number of rows equal to the number of elements of tout, and in each row, the corresponding value of `x`). Note that the name of the function that defines the ODE, i.e., function `logistic_ode`, in the call to `Euler_solver`, does not have to be the same as in function `Euler_solver`, i.e., `odefunction`; rather, the name of the ODE function is specified in the call to `Euler_solver` as `@logistic_ode` where `@` specifies an argument that is a function. This is an important detail since it means that the user of the script `Main_logistic` can select any convenient name for the function that defines the ODEs.

In order to evaluate the numerical solution from `Euler_solver`, the analytical solution (2.8) is evaluated in the function `logistic_exact`.

```
function N = logistic_exact(t)

% Global variables
global a b K N0

% Analytical solution
N = K./(1+(K/N0-1)*exp(-a*t));
```

Function logistic_exact Function to compute the analytical solution of the logistic equation.

The plot produced from the preceding program `Main_logistic` is shown in Fig. 2.1.

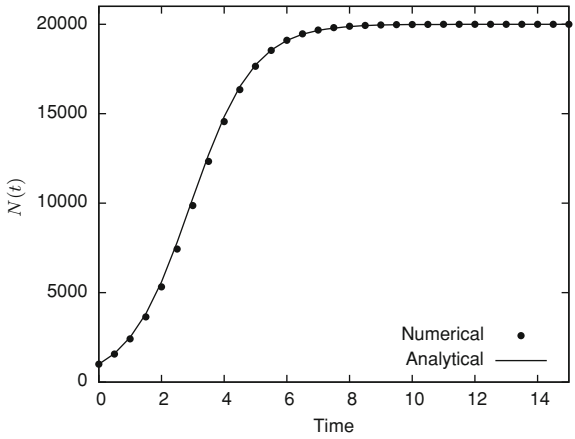

Fig. 2.1 Plot of the numerical and analytical solutions from `Main_logistic`

Note that there is a small difference between the numerical solution (dotted line) and the analytical solution (2.8) (solid line). This difference is due principally to the limited accuracy of the Euler's method and to the small number of Euler steps taken during each output interval (5 steps). Thus, to improve the accuracy of the numerical solution, we could use a more accurate (higher-order) integration algorithm (p refinement discussed in Chap. 1) and/or more integration steps between each output value. We will next consider a higher-order algorithm that generally gives good accuracy with a modest number of integration steps.

2.2 A Basic Variable-Step Nonstiff ODE Integrator

The preceding example illustrates the use of an ODE integrator that proceeds along the solution with a fixed step (constant h). Although this worked satisfactorily for the modest example of Eq. (2.1) (particularly if we used more than five Euler steps in each output interval), we can envision situations where varying the integration step would be advantageous, e.g., the integration step might be relatively small at the beginning of the solution when it changes most rapidly, then increased as the solution changes more slowly. In other words, the integration step will not be maintained at an excessively small value chosen to maintain accuracy where the solution is changing most rapidly, when we would like to use a larger step where the solution is changing less rapidly. The central requirement will then be to vary the integration step so as to maintain a given, specified accuracy.

It might, at first, seem impossible to vary the integration step to maintain a prescribed accuracy since this implies we know the exact solution in order to determine the integration error and thereby decide if the error is under control as the integration step is changed. In other words, if we require the exact solution to determine the integration error, there is no reason to do a numerical integration, e.g., we can just use the exact solution. However, *we can use an estimated error to adjust the integration*

step that does not require knowledge of the exact solution, provided the estimated error is accurate enough to serve as a basis for varying the integration step. Note here the distinction between the exact integration error (generally unknown) and the estimated integration error (generally known).

To investigate the use of an estimated error to adjust the integration step, we will now consider RK methods. Representatives of this class of methods have already been presented in Chap. 1, i.e., the Euler's method, which is a first-order RK method, and the modified Euler or Heun's method, which is a second-order RK method. We are now looking at higher-order schemes, which are based on Taylors series expansions of the solution $x(t)$ of Eq. (1.12), $\frac{\mathrm{d}x}{\mathrm{d}t} = f(x,t)$, i.e.,

$$\begin{aligned} x_{k+1} &= x_k + h \left.\frac{\mathrm{d}x}{\mathrm{d}t}\right|_k + \frac{h^2}{2!} \left.\frac{\mathrm{d}^2x}{\mathrm{d}t^2}\right|_k + \cdots + \frac{h^q}{q!} \left.\frac{\mathrm{d}^q x}{\mathrm{d}t^q}\right|_k + O\left(h^{q+1}\right) \\ &= x_k + hf(x_k,t_k) + \frac{h^2}{2!} \left.\frac{\mathrm{d}f}{\mathrm{d}t}\right|_k + \cdots + \frac{h^q}{q!} \left.\frac{\mathrm{d}^{q-1}f}{\mathrm{d}t^{q-1}}\right|_k + O\left(h^{q+1}\right) \end{aligned} \tag{2.9}$$

The idea behind the RK methods is to evaluate the terms involving higher-order derivatives in (2.9), without actually differentiating the ODEs, but using q intermediate function evaluations $f(x_k,t_k)$ (also called *stages*) between $f(x_k,t_k)$ and $f(x_{k+1},t_{k+1})$, and selecting coefficients w_i so as to match the Taylor series expansion (2.9) with

$$\begin{aligned} x_{k+1} &= x_k + h\sum_{i=1}^{q} w_i f(x_{k,i},t_{k,i}) + O\left(h^{q+1}\right) \\ &= x_k + h\sum_{i=1}^{q} w_i k_i + O\left(h^{q+1}\right) \end{aligned} \tag{2.10}$$

The intermediate stages can generally be expressed as:

$$t_{k,i} = t_k + h\alpha_i, \quad x_{k,i} = x_k + h\sum_{j=1}^{q} \beta_{i,j} k_j \tag{2.11}$$

so that

$$k_i = f\left(x_k + h\sum_{j=1}^{q} \beta_{i,j} k_j, t_k + h\alpha_i\right) \tag{2.12}$$

with $\alpha_1 = 0$ and $\beta_{i,j} = 0\ \forall j \geq i$ an *explicit RK method* is derived (otherwise, the solution of a nonlinear system of Eq. (2.12) is required to get the values of k_i, yielding an *implicit RK method*).

Note that RK methods are *single-step* methods, i.e., x_{k+1} is given in terms of x_k only (in contrast to *multi-step* methods, such as the BDF formulas (1.37), which give

Table 2.1 Maximum order $p_{\max}$ that can be obtained with a q-stage explicit RK method

q	1	2	3	4	5	6	7	8	9
p	1	2	3	4	4	5	6	6	7

x_{k+1} in terms of $x_k, \ldots, x_{k-m}$). They are also called *self starting*, in the sense that only the initial condition x_0 is required to compute x_1.

As a specific example of derivation, consider the explicit, second-order RK methods ($q = 2$), for which Eqs. (2.10)–(2.12) reduce to

$$\begin{aligned} x_{k+1} &= x_k + h\left(w_1 f(x_k, t_k) + w_2 f(x_k + h\beta_{2,1} f(x_k, t_k), t_k + h\alpha_2)\right) \\ &= x_k + hw_1 f(x_k, t_k) + hw_2 \left(f(x_k, t_k) + h\alpha_2 \left.\frac{\partial f}{\partial t}\right|_k + h\beta_{2,1} \left.\frac{\partial f}{\partial x}\right|_k\right) + O\left(h^3\right) \\ &= x_k + h(w_1 + w_2) f(x_k, t_k) + h^2 w_2 \alpha_2 \left.\frac{\partial f}{\partial t}\right|_k \\ &\quad + h^2 w_2 \beta_{2,1} \left.\frac{\partial f}{\partial x}\right|_k f(x_k, t_k) + O\left(h^3\right) \end{aligned} \tag{2.13}$$

Upon comparison with the truncated Taylor series expansion

$$\begin{aligned} x_{k+1} &= x_k + hf(x_k, t_k) + \frac{h^2}{2!} \left.\frac{\mathrm{d}f}{\mathrm{d}t}\right|_k + O\left(h^3\right) \\ &= x_k + hf(x_k, t_k) + \frac{h^2}{2!} \left.\frac{\partial f}{\partial t}\right|_k + \frac{h^2}{2!} \left.\frac{\partial f}{\partial x}\right|_k f(x_k, t_k) + O\left(h^3\right) \end{aligned} \tag{2.14}$$

we obtain

$$w_1 + w_2 = 1, \quad w_2\alpha_2 = \frac{1}{2}, \quad w_2\beta_{2,1} = \frac{1}{2} \tag{2.15}$$

This is a system of three equations and four unknowns which, solved for $w_2 = \lambda$, gives a one-parameter family of explicit methods

$$x_{k+1} = x_k + h\left((1 - \lambda) f(x_k, t_k) + \lambda f\left(x_k + \frac{h}{2\lambda} f(x_k, t_k), t_k + \frac{h}{2\lambda}\right)\right) \tag{2.16}$$

For $\lambda = 0$, we find back the first-order Euler's method (1.16), and for $\lambda = 1/2$, the second-order modified Euler or Heun's method (1.26)–(1.27).

Deriving higher-order RK methods following the same line of thoughts would however require laborious algebraic manipulations. Another approach, based on *graph theory* (*rooted tree theory*), has been proposed by Butcher [6] among others [7, 8]. This approach enables a systematic and an efficient derivation of higher-order explicit and implicit methods. The maximum order $p_{\max}$ that can be obtained with a

q-stage explicit RK method is given in Table 2.1. For a q-stage implicit RK method, $p_{\max} = 2q$. Among the family of fourth- and fifth-order methods, a classical embedded pair of explicit RK formulas is given by

$$x_{k+1,4} = x_k + \frac{25}{216}k_1 + \frac{1408}{2565}k_3 + \frac{2197}{4104}k_4 - \frac{1}{5}k_5 \tag{2.17}$$

$$x_{k+1,5} = x_k + \frac{16}{135}k_1 + \frac{6656}{12825}k_3 + \frac{28561}{56430}k_4 - \frac{9}{50}k_5 + \frac{2}{55}k_6 \tag{2.18}$$

These formulas are said *embedded* as they share the same stages k_i, $i = 1, \ldots, 5$. Such stages can be computed as follows:

$$k_1 = f(x_k, t_k)h \tag{2.19a}$$

$$k_2 = f\left(x_k + 0.25k_1, t_k + 0.25h\right)h \tag{2.19b}$$

$$k_3 = f\left(x_k + \frac{3}{32}k_1 + \frac{9}{32}k_2, t_k + \frac{3}{8}h\right)h \tag{2.19c}$$

$$k_4 = f\left(x_k + \frac{1932}{2197}k_1 - \frac{7200}{2197}k_2 + \frac{7296}{2197}k_3, t_k + \frac{12}{13}h\right)h \tag{2.19d}$$

$$k_5 = f\left(x_k + \frac{439}{216}k_1 - 8k_2 + \frac{3680}{513}k_3 - \frac{845}{4104}k_4, t_k + h\right)h \tag{2.19e}$$

$$k_6 = f\left(x_k - \frac{8}{27}k_1 + 2k_2 - \frac{3544}{2565}k_3 + \frac{1859}{4104}k_4 - \frac{11}{40}k_5, t_k + 0.5h\right)h \tag{2.19f}$$

Returning to the idea of using an estimated error to adjust the integration step, we can obtain an error estimate for the fourth-order result from

$$\varepsilon_{k+1,4} = x_{k+1,5} - x_{k+1,4} \tag{2.20}$$

that can be then used to adjust the integration step h in accordance with a user-prescribed error tolerance.

Equation (2.17) fits the underlying Taylor series up to including the fourth-order term $\frac{\mathrm{d}^4 x_k}{\mathrm{d}t^4}\left(\frac{h^4}{4!}\right)$ while Eq. (2.18) fits the Taylor series up to an including the fifth-order term $\frac{\mathrm{d}^5 x_k}{\mathrm{d}t^5}\left(\frac{h^5}{5!}\right)$. Thus, the subtraction (2.20) provides an estimate of the fifth-order term.

The RK pair (2.17)–(2.18) is termed the *Runge–Kutta–Fehlberg method* [9], usually designated *RKF45*. Note that an essential feature of an embedded pair as illustrated by Eq. (2.19) is that the stages are the same for the lower and higher-order methods (k_1–k_6 in this case). Therefore, the subtraction (2.20) can be used

to combine terms of the same stages (since they are the same for the lower- and higher-order methods and therefore have to be *calculated only once for both of them*).

Two functions for implementing the *RKF45* method, with the use of Eq. (2.20) to estimate the truncation error and adjust the integration step, are `ssrkf45` and `rkf45_solver` listed below. The first function, `ssrkf45`, takes a single step along the solution based on Eqs. (2.17)–(2.18):

1. After a block of comments explaining the operation of `ssrkf45`, the derivative vector is computed at the base point and the first stage, k_1, is evaluated according to (2.19a). Note that k_1 is a column vector with the row dimension equal to the number of first order ODEs to be integrated. Each stage is just the ODE derivative vector multiplied by the integration step h, so each `xt0` is calculated by a call to the derivative routine via `feval`.
2. The dependent variable vector and the independent variable are then evaluated to initiate the calculation of the next stage, k_2, according to (2.19b).
3. This basic procedure is repeated for the calculation of k_3, k_4, k_5 and k_6 according to (2.19c)–(2.19f). Then the fourth- and fifth-order solutions are computed at the next point by the application of Eqs. (2.17)–(2.18), and the error is estimated according to (2.20).
4. The independent variable, the fifth-order solution and the vector of estimated errors at the next point along the solution are then returned from `ssrkf45` as `t`, `x`, and `e` (the output or return arguments of `ssrkf45`).

```
function [t,x,e] = ssrkf45(odefunction,t0,x0,h)
%
% This function computes an ODE solution by the RK Fehlberg 45
% method for one step along the solution (by calls to
% 'odefunction' to define the ODE derivative vector).  It also
% estimates the truncation error of the solution, and applies
% this estimate as a correction to the solution vector.
%
% Argument list
%
% odefunction – string containing name of user–supplied problem
%   t0 – initial value of independent variable
%   x0 – initial condition vector
%   h  – integration step
%   t  – independent variable (scalar)
%   x  – solution vector after one rkf45 step
%   e  – estimate of truncation error of the solution vector

% Derivative vector at initial (base) point
[xt0] = feval(odefunction,t0,x0);

% k1, advance of dependent variable vector and independent
% variable for calculation of k2
k1 = h*xt0;
x = x0 + 0.25*k1;
t = t0 + 0.25*h;

% Derivative vector at new x, t
```

```
[xt] = feval(odefunction,t,x);

% k2, advance of dependent variable vector and independent
% variable for calculation of k3
k2 = h*xt;
x = x0 + (3.0/32.0)*k1...
       + (9.0/32.0)*k2;
t = t0 + (3.0/8.0)*h;

% Derivative vector at new x, t
[xt] = feval(odefunction,t,x);

% k3, advance of dependent variable vector and independent
% variable for calculation of k4
k3 = h*xt;
x = x0 + (1932.0/2197.0)*k1...
    - (7200.0/2197.0)*k2...
    + (7296.0/2197.0)*k3;
t = t0 + (12.00/13.0)*h;

% Derivative vector at new x, t
[xt] = feval(odefunction,t,x);

% k4, advance of dependent variable vector and independent
% variable for calculation of k5
k4 = h*xt;
x = x0 + ( 439.0/ 216.0)*k1...
    - (   8.0       )*k2...
    + (3680.0/ 513.0)*k3...
    - ( 845.0/4104.0)*k4;
t = t0 + h;

% Derivative vector at new x, t
[xt] = feval(odefunction,t,x);

% k5, advance of dependent variable vector and independent
% variable for calculation of k6
k5 = h*xt;
x = x0 - (   8.0/  27.0)*k1...
    + (   2.0       )*k2...
    - (3544.0/2565.0)*k3...
    + (1859.0/4104.0)*k4...
    - (  11.0/  40.0)*k5;
t = t0 + 0.5*h;

% Derivative vector at new u, t
[xt] = feval(odefunction,t,x);

% k6
k6 = h*xt;

% Fourth order step
sum4 = x0 + (    25.0/ 216.0)*k1...
          + (  1408.0/2565.0)*k3...
          + (  2197.0/4104.0)*k4...
          - (     1.0/   5.0)*k5;

% Fifth order step
sum5 = x0 + (   16.0/  135.0)*k1...
          + ( 6656.0/12825.0)*k3...
          + (28561.0/56430.0)*k4...
          - (    9.0/   50.0)*k5...
          + (    2.0/   55.0)*k6;
t = t0 + h;

% Truncation error estimate
```

```
e = sum5 - sum4;

% Fifth order solution vector (from 4,5 RK pair);
% two ways to the same result are listed
% x = sum4 + e;
x = sum5;
```

Function ssrkf45 Function for a single step along the solution of (1.12) based on (2.15)–(2.17)

We also require a function that calls `ssrkf45` to take a series of step along the solution. This function, `rkf45_solver`, is described below:

1. After an initial block of comments explaining the operation of `rkf45_solver` and listing its input and output arguments, the integration is initialized. In particular, the initial integration step is set to `h = Dtplot/2` and the independent variable `tout`, the dependent variable vector `xout` and the estimated error vector `eout` are initiated for subsequent output. The integration steps, `nsteps`, and the number of outputs, `nplots`, are also initialized.
2. Two loops are then executed. The outer loop steps through `nplots` outputs. Within each pass through this outer loop, the integration is continued while the independent variable `t` is less than the final value `tplot` for the current output interval. Before entering the inner `while` loop, the length of the output interval, `tplot`, is set.
3. Within each of the output intervals, a logic variable is initialized to specify a successful integration step and a check is made to determine if the integration step, `h`, should be reset to cover the remaining distance in the output interval.
4. `ssrkf45` is then called for one integration step of length `h`.
5. A series of tests then determines if the integration interval should be changed. First, if the estimated error (for any dependent variable) exceeds the error tolerances (note the use of a combination of the absolute and relative error tolerances), the integration step is halved. If the integration step is reduced, the logic variable `fin1` is set to 0 so that the integration step is repeated from the current base point.
6. Next (for `fin1 = 1`), if the estimated error for all of the dependent variables is less than $1/32$ of the composite error tolerance, the step is doubled. The factor $1/32$ is used in accordance with the fifth-order RK algorithm. Thus, if the integration step is doubled, the integration error will increase by a factor of $2^5 = 32$. If the estimated error for *any* dependent variable exceeds $1/32$ of the composite error tolerance, the integration step is unchanged (`fin1 = 0`).
7. Next, two checks are made to determine if user-specified limits have been reached. If the integration step has reached the specified minimum value, the integration interval is set to this minimum value and the integration continues from this point. If the maximum number of integration steps has been exceeded, an error message is displayed, execution of the `while` and `for` loops is terminated through the two `breaks`, i.e., the ODE integration is terminated since the total number of integration steps has reached the maximum value.

8. Finally, at the end of the output interval, the solution is stored in arrays, which are the return arguments for `rkf45_solver`. Note that the final `end` statement terminates the `for` loop that steps the integration through the `nplots` outputs.

```
function [tout,xout,eout]=rkf45_solver(odefunction,t0,tf,x0,...
                                       hmin,nstepsmax,abstol,...
                                       reltol,Dtplot)
% This function solves first-order differential equations using
% the Runga-Kutta-Felhberg (4,5) method
% [tout, xout] = rkf45_solver(@f,t0,tf,x0,hmin,nstepsmax,...
%                             abstol,reltol,Dtplot)
% integrates the system of differential equations xt=f(t,x)
% from t0 to tf with initial conditions x0.  f is a string
% containing the name of an ODE file.  Function f(t,x) must
% return a column vector. Each row in solution array xout
% corresponds to a value returned in column vector t.
%
% rkf45_solver.m solves first-order differential equations
% using the variable step RK Fehlberg 45 method for a series of
% points along the solution by repeatedly calling function
% ssrkf45 for a single RK Fehlberg 45 step.  The truncation error
% is estimated along the solution to adjust the integration step
% according to a specified error tolerance.
%
% Argument list
%
% f - String containing name of user-supplied problem description
%       Call: xt = problem_name(t,x) where f = 'problem_name'
%             t  - independent variable (scalar)
%             x  - solution vector
%             xt - returned derivative vector; xt(i) = dx(i)/dt
%
% t0        - initial value of t
% tf        - final value of t
% x0        - initial value vector
% hmin      - minimum allowable time step
% nstepsmax - maximum number of steps
% abstol    - absolute error tolerance
% reltol    - relative error tolerance
% Dtplot    - plot interval
% tout      - returned integration points (column-vector)
% xout      - returned solution, one solution row-vector per
%             tout-value

% Start integration
t = t0;
tini = t0;
tout = t0;                          % initialize output value
xout = x0';                         % initialize output value
eout = zeros(size(xout));           % initialize output value
nsteps = 0;                         % initialize step counter
nplots = round((tf-t0)/Dtplot);     % number of outputs

% Initial integration step
h = 10*hmin;

% Step through nplots output points
for i = 1:nplots
    % Final (output) value of the independent variable
    tplot = tini+i*Dtplot;
    % While independent variable is less than the final value,
    % continue the integration
```

```
    while t <= tplot*0.9999
        % If the next step along the solution will go past the
        % final value of the independent variable, set the step
        % to the remaining distance to the final value
        if t+h > tplot, h = tplot-t; end

        % Single rkf45 step
        [t,x,e] = ssrkf45(odefunction,t0,x0,h);

        % Check if any of the ODEs have violated the error
        % criteria
        if max( abs(e) > (abs(x)*reltol + abstol) )
            % Error violation, so integration is not complete.
            % Reduce integration step because of error violation
            % and repeat integration from the base point.
            % Set logic variable for rejected integration step.
            h = h/2;

            % If the current step is less than the minimum
            % allowable step, set the step to the minimum
            % allowable value
            if h < hmin,  h = hmin; end

            % If there is no error violation, check if there is
            % enough "error margin" to increase the integration
            % step
        elseif max( abs(e) > (abs(x)*reltol + abstol)/32 )
            % The integration step cannot be increased, so leave
            % it unchanged and continue the integration from the
            % new base point.
            x0 = x; t0 = t;
            % There is no error violation and enough "security
            % margin"
        else
            % double integration step and continue integration
            % from new base point
            h = 2*h; x0 = x; t0 = t;
        end %if

        % Continue while and check total number of integration
        % steps taken
        nsteps=nsteps+1;
        if(nsteps > nstepsmax)
     fprintf(' \n nstepsmax exceeded; integration terminated\n');
     break;
        end
    end %while

    % add latest result to the output arrays and continue for
    % loop
    tout = [tout ; t];
    xout = [xout ; x'];
    eout = [eout ; e'];
end % for
% End of rkf45_solver
```

Function rkf45_solver Function for a variable step solution of an ODE system

In summary, we now have provision for increasing, decreasing or not changing the integration interval in accordance with a comparison of the estimated integration error (for each dependent variable) with the composite error tolerance, and for taking some

special action if user-specified limits (minimum integration interval and maximum number of integration steps) are exceeded.

We can now apply `ssrkf45` an `rkf45_solver` to the logistic equation (2.1). We already have function `logistic_ode` and the exact solution in `logistic_exact`, so all we now require is a main program, which closely parallels the main program calling the fixed step Euler integrator (`Main_logistic`). The complete code is available in the companion library. Here, we just note a few points:

1. Since `rkf45_solver` is a variable step integrator, it requires error tolerances, a minimum integration step and the maximum number of steps

```
abstol    = 1e-3;
reltol    = 1e-3;
hmin      = 1e-3;
nstepsmax = 1000;
```

   These parameters are inputs to `rkf45_solver`.

```
% Call to ODE solver
[tout,xout] = rkf45_solver(@logistic_ode,t0,tf,N0,...
                           hmin,nstepsmax,abstol,...
                           reltol,Dtplot);
```

2. In order to assess the accuracy of the numerical solution, the absolute and relative errors are computed from the exact solution. These (exact) errors can then be compared with the absolute and relative error tolerances

```
% Print results
fprintf('   t    x(t)   xex(t)   abserr    relerr\n');
for i = 1:length(tout)
    fprintf('%7.1f%10.2f%10.2f%10.5f%10.5f\n',...
     tout(i),xout(i,1),Nexact(i),xout(i,1)-Nexact(i),...
      (xout(i,1)-Nexact(i))/Nexact(i));
end
```

The plot produced by the main program is shown in Fig. 2.2. The exact and numerical solutions are indistinguishable (as compared with Fig. 2.1). This issue is also confirmed by the numerical output reproduced in Table 2.2.

We can note the following features of the numerical solution:

1. The initial conditions of the numerical and exact solutions agree (a good check).
2. The maximum relative error is -0.00001 which is a factor of 0.01 better than the relative error tolerance set in the main program, i.e., `reltol = 1e-3;`
3. The maximum absolute error is -0.02847 which exceeds the absolute error tolerance, i.e., `abstol = 1e-3;` this absolute error tolerance can be considered excessively stringent since it specifies 0.001 for dependent variable values between 1,000 ($t = 0$) and 20,000 ($t = 15$). To explain why this absolute error tolerance is not observed in the numerical solution, consider the use of the absolute and relative error tolerances in function `rkf45_solver`, i.e., `if`

Table 2.2 Numerical output obtained with `rkf45_solver` (`abstol = 1e-3`)

t	x(t)	xex(t)	abserr	relerr
0.0	1000.00	1000.00	0.00000	0.00000
0.5	1596.92	1596.92	-0.00033	-0.00000
1.0	2503.21	2503.22	-0.00958	-0.00000
1.5	3817.16	3817.18	-0.01991	-0.00001
2.0	5600.06	5600.09	-0.02673	-0.00000
2.5	7813.65	7813.68	-0.02799	-0.00000
3.0	10277.71	10277.73	-0.02742	-0.00000
3.5	12708.47	12708.50	-0.02847	-0.00000
4.0	14836.80	14836.83	-0.02623	-0.00000
4.5	16514.30	16514.31	-0.01358	-0.00000
5.0	17730.17	17730.17	0.00552	0.00000
5.5	18558.95	18558.92	0.02088	0.00000
6.0	19100.47	19100.44	0.02777	0.00000
6.5	19444.59	19444.56	0.02766	0.00000
7.0	19659.41	19659.39	0.02385	0.00000
7.5	19792.03	19792.01	0.01888	0.00000
8.0	19873.35	19873.33	0.01416	0.00000
8.5	19922.99	19922.98	0.01023	0.00000
9.0	19953.22	19953.21	0.00720	0.00000
9.5	19971.60	19971.60	0.00497	0.00000
10.0	19982.77	19982.76	0.00338	0.00000
10.5	19989.54	19989.54	0.00227	0.00000
11.0	19993.66	19993.66	0.00151	0.00000
11.5	19996.15	19996.15	0.00100	0.00000
12.0	19997.67	19997.67	0.00065	0.00000
12.5	19998.58	19998.58	0.00043	0.00000
13.0	19999.14	19999.14	0.00028	0.00000
13.5	19999.48	19999.48	0.00018	0.00000
14.0	19999.68	19999.68	0.00012	0.00000
14.5	19999.81	19999.81	0.00007	0.00000
15.0	19999.88	19999.88	0.00005	0.00000

Table 2.3 Numerical output obtained with `rkf45_solver` (`abstol=1`)

t	x(t)	xex(t)	abserr	relerr
0.0	1000.00	1000.00	0.00000	0.00000
0.5	1596.92	1596.92	-0.00033	-0.00000
1.0	2503.21	2503.22	-0.00958	-0.00000
1.5	3817.16	3817.18	-0.01991	-0.00001
2.0	5600.06	5600.09	-0.02673	-0.00000
2.5	7813.65	7813.68	-0.02799	-0.00000
3.0	10277.71	10277.73	-0.02742	-0.00000
3.5	12708.47	12708.50	-0.02847	-0.00000
4.0	14836.80	14836.83	-0.02623	-0.00000
4.5	16514.30	16514.31	-0.01358	-0.00000
5.0	17730.17	17730.17	0.00552	0.00000

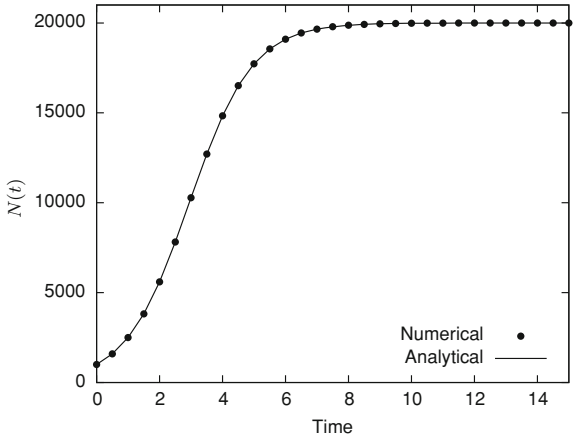

Fig. 2.2 Plot of the numerical (using the `rkf45` IVP solver) and analytical solutions of the logistic equation

`abs(e(i)) > (abs(x(i))*reltol + abstol)`. This composite error is typically `(1,000)*1.0e-03 + 1.0e-03` and therefore the contribution of the absolute error tolerance (the second `1.0e-03`) is small in comparison to the contribution of the relative error tolerance (the first `1.0e-03`). In other words, the relative error tolerance controls the integration step in this case, which explains why the absolute error tolerance in the output of Table 2.3 exceeds the absolute error tolerance. This conclusion does emphasize the need to carefully understand and specify the error tolerances.

To explore this idea a little further, if the absolute error tolerance had been specified as `abserr = 1`, it would then be consistent with the relative error, i.e., both the absolute error (=1) and the relative error of `1.0e-03` are 1 part in 1,000 for the dependent variable `N` equal to its initial value of `1,000`.

A sample of the output is given in Table 2.3. Note that this output is essentially identical to that of Table 2.2 (thus confirming the idea that the absolute error tolerance `1.0e-03` has essentially no effect on the numerical solution), but now both error criteria are satisfied. Specifically, the maximum absolute error −0.02847 is well below the specified absolute error (=1).

This example illustrates that the specification of the error tolerances requires some thought, including the use of representative values of the dependent variable, in this case `N = 1,000` for deciding on appropriate error tolerances. Also, this example brings to mind the possibility that the dependent variable may have a value of zero in which case the relative error tolerance has no effect in the statement `if abs(e(i)) > (abs(x(i))*reltol + abstol)` (i.e., `x(i) = 0`) and therefore the absolute error tolerance completely controls the step changing algorithm. In other words, specifying an absolute error tolerance of zero may not be a good idea (if the dependent variable passes through a zero value).

This situation of some dependent variables passing through zero also suggests that having absolute and relative error tolerances that might be different for each dependent variable might be worthwhile, particularly if the dependent

variables have typical values that are widely different. In fact, this idea is easy to implement since the integration algorithm returns an estimated integration error for each dependent variable as a vector (`e(i)` in the preceding program statement). Then, if absolute and relative error tolerances are specified as vectors (e.g., `reltol(i)` and `abstol(i)`), the comparison of the estimated error with the absolute and relative error tolerances for each dependent variable is easily accomplished, e.g., by using a `for` loop with index `i`. Some library ODE integrators have this feature (of specifying absolute and relative error vectors), and it could easily be added to `rkf45_solver`, for example. Also, the idea of a relative error brings widely different values of the dependent variables together on a common scale, while the absolute error tolerance should reflect these widely differing values, particularly for the situation when some of the dependent variables pass through zero.

4. As a related point, we have observed that the error estimator for the *RKF45* algorithm (i.e., the difference between the fourth- and fifth-order solutions) is conservative in the sense that it provides estimates that are substantially above the actual (exact) error. This is desirable since the overestimate of the error means that the integration step will be smaller than necessary to achieve the required accuracy in the solution as specified by the error tolerances. Of course, if the error estimate is too conservative, the integration step might be excessively small, but this is better than an error estimate that is not conservative (too small) and thereby allows the integration step to become so large the actual error is above the error tolerances. In other words, we want an error estimate that is reliable.
5. Returning to the output of Table 2.2, the errors in the numerical solution actually decrease after reaching maximum values. This is rather typical and very fortuitous, i.e., the errors do not accumulate as the solution proceeds.
6. The numerical solution approaches the correct final value of 20,000 (again, this is important in the sense that the errors do not cause the solution to approach an incorrect final value).

In summary, we now have an *RKF45* integrator that can be applied to a broad spectrum of initial value ODE problems. In the case of the preceding application to the logistic equation, *RKF45* was sufficiently accurate that only one integration step was required to cover the entire output interval of 0.5 (this was determined by putting some additional output statements in `rkf45_solver` to observe the integration step h, which illustrates an advantage of the basic integrators, i.e., experimentation with supplemental output is easily accomplished). Thus, the integration step adjustment in `rkf45_solver` was not really tested by the application to the logistic equation.

Therefore, we next consider another application which does require that the integration step is adjusted. At the same time with this application, we will investigate the notion of stiffness. Indeed, there is one important limitation of *RKF45*: it is an explicit integrator which does not perform well when applied to stiff problems.

The next application consists of a system of two linear, constant coefficient ODEs

$$\frac{dx_1}{dt} = -ax_1 + bx_2 \tag{2.21a}$$

$$\frac{dx_2}{dt} = bx_1 - ax_2 \tag{2.21b}$$

For the initial conditions

$$x_1(0) = 0; \quad x_2(0) = 2 \tag{2.22}$$

the analytical solution to Eq. (2.21) is

$$x_1(t) = e^{\lambda_1 t} - e^{\lambda_2 t} \tag{2.23a}$$

$$x_2(t) = e^{\lambda_1 t} + e^{\lambda_2 t} \tag{2.23b}$$

where

$$\lambda_1 = -(a-b); \quad \lambda_2 = -(a+b) \tag{2.24}$$

and a, b are constants to be specified.

We again use the *RKF45* algorithm implemented in functions `rkf45_solver` and `ssrkf45`. The main program, designated `stiff_main`, and associated functions `stiff_odes` and `stiff_odes_exact` (which compute the exact solution from Eq. (2.23) to (2.24)) can be found in the companion library and follows directly from the previous discussion. Note that the absolute and relative error tolerances are chosen as `abstol=1e-4` and `reltol=1e-4`, which is appropriate for the two dependent variables $x_1(t)$ and $x_2(t)$ since they have representative values of 1. Further, since $x_1(t)$ has an initial condition of zero, the specification of an absolute error tolerance is essential.

The numerical output from these functions is listed in abbreviated form in Table 2.4. As we can see in the table, two rows are printed at each time instant. The first row corresponds with the output for the first state variable (x_1) while the second row corresponds with the second state variable (x_2). The error tolerances are easily satisfied throughout this solution and the number of steps taken by the solver is `nsteps = 24`. The plotted solution is shown in Fig. 2.3. The eigenvalues for this solution with $a = 1.5$, $b = 0.5$ are $\lambda_1 = -(a-b) = -1$ and $\lambda_2 = -(a+b) = -2$. As t increases, e^{-2t} decays more rapidly than e^{-t} and eventually becomes negligibly small (in comparison to e^{-t}). Therefore, from Eq. (2.21), the two solutions merge (at about $t = 5$ from Fig. 2.3). In other words, the solution for both $x_1(t)$ and $x_2(t)$ becomes essentially e^{-t} for $t > 5$.

We can now vary a and b to investigate how these features of the solution change and affect the numerical solution. For example, if $a = 10.5$, $b = 9.5$, the eigenvalues are $\lambda_1 = -1$ and $\lambda_2 = -20$ so that the exponential $e^{\lambda_2 t} = e^{-20t}$ decays much more rapidly than $e^{\lambda_1 t} = e^{-t}$. The corresponding plotted solution is shown in Fig. 2.4. Note that the two solutions merge at about $t = 0.5$. Also, the specified error criteria, `abserr = 1.0e-04`, `relerr=1.0e-04`, are satisfied throughout the solution, but clearly, this is becoming more difficult for the variable-step algorithm to accomplish because of the rapid change in the solution just after the initial condition.

Table 2.4 Numerical output from `stiff_main` for $a = 1.5$, $b = 0.5$

t	x(t)	xex(t)	abserr	relerr
0.00	0.0000000	0.0000000	0.0000000	0.0000000
	2.0000000	2.0000000	0.0000000	0.0000000
0.10	0.0861067	0.0861067	0.0000000	0.0000000
	1.7235682	1.7235682	-0.0000000	-0.0000000
0.20	0.1484108	0.1484107	0.0000000	0.0000003
	1.4890508	1.4890508	-0.0000000	-0.0000000
0.30	0.1920067	0.1920066	0.0000001	0.0000004
	1.2896298	1.2896299	-0.0000001	-0.0000001
0.40	0.2209912	0.2209911	0.0000001	0.0000004
	1.1196489	1.1196490	-0.0000001	-0.0000001
0.50	0.2386513	0.2386512	0.0000001	0.0000004
	0.9744100	0.9744101	-0.0000001	-0.0000001
⋮	⋮	⋮	⋮	⋮
9.60	0.0000677	0.0000677	-0.0000000	-0.0000001
	0.0000677	0.0000677	-0.0000000	-0.0000001
9.70	0.0000613	0.0000613	-0.0000000	-0.0000001
	0.0000613	0.0000613	-0.0000000	-0.0000001
9.80	0.0000554	0.0000554	-0.0000000	-0.0000001
	0.0000555	0.0000555	-0.0000000	-0.0000001
9.90	0.0000502	0.0000502	-0.0000000	-0.0000001
	0.0000502	0.0000502	-0.0000000	-0.0000001
10.00	0.0000454	0.0000454	-0.0000000	-0.0000001
	0.0000454	0.0000454	-0.0000000	-0.0000001

Practically, this manifests in an increased number of integration steps required to meet the specified error tolerances, i.e., `nsteps = 78`.

If this process of separating the eigenvalues is continued, clearly the difficulty in computing a numerical solution will increase, due to two causes:

- The initial "transient" (or "boundary layer") in the solution just after the initial condition will become shorter and therefore more difficult for the variable step algorithm to resolve.
- As the eigenvalues separate, the problem becomes stiffer and the stability limit of the explicit *RKF45* integrator places an even smaller limit on the integration step to maintain stability. The maximum allowable integration step to maintain stability can be estimated from the approximate stability condition for *RKF45* (see Fig. 1.6, in particular the curve corresponding to the fourth-order method)

$$|\lambda h| < 2.7 \tag{2.25}$$

Thus, for $\lambda_2 = -20$, $h < \frac{2.7}{20} \approx 0.135$ which is still not very restrictive (relative to the time scale of $0 \leq t \leq 5$ in Fig. 2.4) so that for this case, accuracy still probably dictates the maximum integration step (to meet the error tolerances).

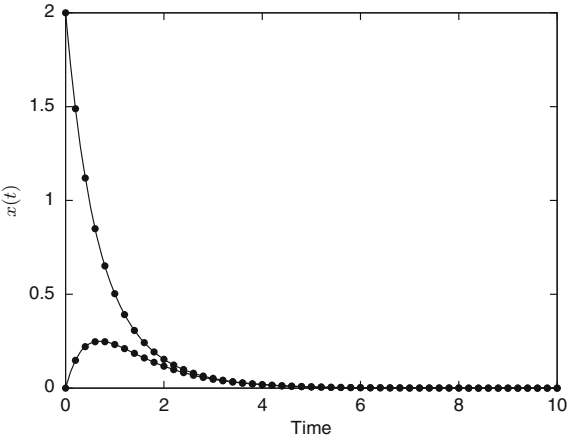

Fig. 2.3 Plot of the numerical and analytical solutions (2.21)–(2.24) with $a = 1.5, b = 0.5$

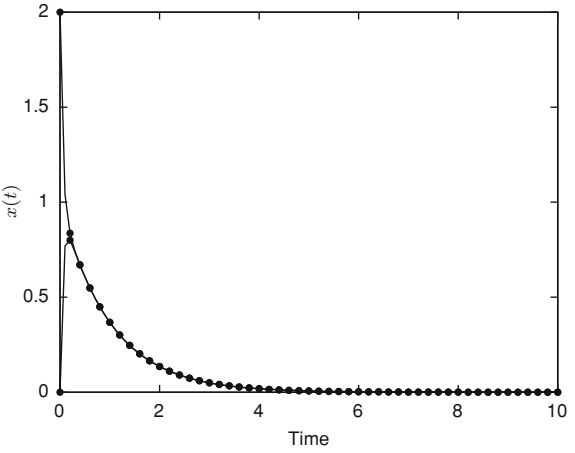

Fig. 2.4 Plot of the numerical and analytical solutions (2.21)–(2.24) with $a = 10.5, b = 9.5$

We consider one more case in separating the eigenvalues, $a = 5000.5, b = 4999.5$, for which $\lambda_1 = -1, \lambda_2 = -10{,}000$; thus, the *stiffness ratio* is 10,000/1. The output for this case is incomplete since `rkf45_solver` fails at $t = 0.33$ with an error message indicating nsteps has exceeded the limit `nstepsmax=1,000`. This is to be expected since the maximum step size to still maintain stability is now determined by $\lambda_2 = -10{,}000$, $h < \frac{2.7}{10000} = 2.7 \times 10^{-4}$. In other words, to cover the total time interval $0 \le t \le 5$, which is set by $\lambda_1 = -1$ (so that the exponential for λ_1 decays to $e^{-(1)(5)}$), the integrator must take at least $\frac{5}{2.7 \times 10^{-4}} \approx 2 \times 10^4$ steps which is greater than `nstepsmax=1,000` (i.e., the numerical solution only proceeded to $t = 0.33$ when `nstepsmax=1,000` was exceeded).

Note in general that a stiff system (with widely separated eigenvalues) has the characteristic that the total time scale is determined by the smallest eigenvalue (e.g., $\lambda_1 = -1$) while the maximum step allowed to still maintain stability in covering

this total time interval is determined by the largest eigenvalue (e.g., $\lambda_2 = -10{,}000$); if these two extreme eigenvalues are widely separated, the combination makes for a large number of integration steps to cover the total interval (again, as demonstrated by *RKF45* which in this case could only get to $t = 0.33$ with 10,00 steps).

The preceding example illustrates the general limitation of the stability of *explicit ODE integrators* for the solution of stiff systems. We therefore now consider *implicit ODE integrators*, which generally circumvent this stability limit.

2.3 A Basic Variable Step Implicit ODE Integrator

Many implicit ODE integrators have been proposed and implemented in computer codes. Thus, to keep the discussion to a reasonable length, we consider here only one class of methods to illustrate some properties that clearly demonstrate the advantages of using an implicit integrator. This type of integrator is generally termed as *Rosenbrock* [10] or *linearly implicit Runge–Kutta (LIRK)* method.

Consider an autonomous function f (f does not depend explicitly on time) and the following equation:

$$\frac{\mathrm{d}x}{\mathrm{d}t} = f(x) \tag{2.26}$$

The explicit RK methods developed earlier in this chapter (see Eqs. 2.10–2.12) are given by

$$\begin{aligned}
k_1 &= f(x_k) \\
k_2 &= f(x_k + h\beta_{2,1}k_1) \\
k_3 &= f(x_k + h(\beta_{3,1}k_1 + \beta_{3,2}k_2)) \\
&\vdots \\
k_q &= f(x_k + h(\beta_{q,1}k_1 + \beta_{q,2}k_2 + \cdots + \beta_{q,q-1}k_{q-1})) \\
x_{k+1} &= x_k + h\sum_{i=1}^{q} w_i k_i
\end{aligned} \tag{2.27}$$

whereas the general implicit formulas can be expressed by

$$\begin{aligned}
k_1 &= f(x_k + h(\beta_{1,1}k_1 + \beta_{1,2}k_2 + \cdots + \beta_{1,q}k_q)) \\
k_2 &= f(x_k + h(\beta_{2,1}k_1 + \beta_{2,2}k_2 + \cdots + \beta_{2,q}k_q)) \\
k_3 &= f(x_k + h(\beta_{3,1}k_1 + \beta_{3,2}k_2 + \cdots + \beta_{3,q}k_q)) \\
&\vdots \\
k_q &= f(x_k + h(\beta_{q,1}k_1 + \beta_{q,2}k_2 + \cdots + \beta_{q,q}k_q))
\end{aligned} \tag{2.28}$$

$$x_{k+1} = x_k + h \sum_{i=1}^{q} w_i k_i$$

Diagonally implicit Runge–Kutta (DIRK) is a particular case of this general formulation where $\beta_{i,j} = 0$ for $j > i$, i.e.,

$$\begin{aligned} k_1 &= f(x_k + h\beta_{1,1}k_1) \\ k_2 &= f(x_k + h(\beta_{2,1}k_1 + \beta_{2,2}k_2)) \\ k_3 &= f(x_k + h(\beta_{3,1}k_1 + \beta_{3,2}k_2 + \beta_{3,3}k_3)) \\ &\vdots \\ k_q &= f(x_k + h(\beta_{q,1}k_1 + \beta_{q,2}k_2 + \cdots + \beta_{q,q}k_q)) \\ x_{k+1} &= x_k + h \sum_{i=1}^{q} w_i k_i \end{aligned} \tag{2.29}$$

LIRK methods introduce a linearization of stage k_i

$$\begin{aligned} k_i &= f(x_k + h(\beta_{i,1}k_1 + \cdots + \beta_{i,i-1}k_{i-1}) + h\beta_{i,i}k_i) \\ k_i &\approx f(x_k + h(\beta_{i,1}k_1 + \cdots + \beta_{i,i-1}k_{i-1})) \\ &\quad + h\beta_{i,i} \left.\frac{\partial f}{\partial x}\right|_{x_k + h(\beta_{i,1}k_1 + \cdots + \beta_{i,i-1}k_{i-1})} k_i \end{aligned} \tag{2.30}$$

or, if we consider a system of ODEs $\frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t} = \mathbf{f}(\mathbf{x})$ (instead of a single ODE)

$$\begin{aligned} \mathbf{k}_i &= \mathbf{f}(\mathbf{x}_k + h(\beta_{i,1}\mathbf{k}_1 + \cdots + \beta_{i,i-1}\mathbf{k}_{i-1})) \\ &\quad + h\beta_{i,i} \left.\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\right|_{\mathbf{x}_k + h(\beta_{i,1}\mathbf{k}_1 + \cdots + \beta_{i,i-1}\mathbf{k}_{i-1})} \mathbf{k}_i \end{aligned} \tag{2.31}$$

where the Jacobian $\left.\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\right|_{\mathbf{x}_k + h(\beta_{i,1}\mathbf{k}_1 + \cdots + \beta_{i,i-1}\mathbf{k}_{i-1})} \approx \left.\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\right|_{\mathbf{x}_k}$ is usually not evaluated for each stage, but rather assumed constant across the stages, so as to keep the computational expense at a reasonable level.

In addition, Rosenbrock's methods replace stage $\mathbf{k}_i$ in the preceding expression by a linear combination of the previous stages (constructed so as to preserve the lower-triangular structure, i.e., $\gamma_{i,j} = 0$ for $j > i$)

$$\begin{aligned} \mathbf{k}_1 &= \mathbf{f}(\mathbf{x}_k) + h\beta_{1,1}\mathbf{J}_k\mathbf{k}_1 \\ \mathbf{k}_2 &= \mathbf{f}(\mathbf{x}_k + h\beta_{2,1}\mathbf{k}_1) + h\mathbf{J}_k\left(\gamma_{2,1}\mathbf{k}_1 + \gamma_{2,2}\mathbf{k}_2\right) \\ &\vdots \\ \mathbf{k}_q &= \mathbf{f}(\mathbf{x}_k + h(\beta_{q,1}\mathbf{k}_1 + \beta_{q,2}\mathbf{k}_2 + \cdots + \beta_{q,q-1}\mathbf{k}_{q-1})) \end{aligned} \tag{2.32}$$

$$
\begin{aligned}
&+ h\mathbf{J}_k \left(\gamma_{2,1}\mathbf{k}_1 + \cdots + \gamma_{q,q}\mathbf{k}_q\right) \\
\mathbf{x}_{k+1} &= \mathbf{x}_k + h\sum_{i=1}^{q} w_i \mathbf{k}_i
\end{aligned}
$$

Therefore, each stage $\mathbf{k}_i$ can be computed by solving a linear system of equations of the form

$$
\begin{aligned}
\left(\mathbf{I} - h\gamma_{i,i}\mathbf{J}_k\right)\mathbf{k}_i = &\mathbf{f}\left(\mathbf{x}_k + h(\beta_{i,1}\mathbf{k}_1 + \beta_{i,2}\mathbf{k}_2 + \cdots + \beta_{i,i-1}\mathbf{k}_{i-1})\right) \\
&+ h\mathbf{J}_k \left(\gamma_{i,1}\mathbf{k}_1 + \cdots + \gamma_{i,i-1}\mathbf{k}_{i-1}\right)
\end{aligned} \tag{2.33}
$$

If the parameters $\gamma_{i,i}$ are all given the same numerical values

$$
\gamma_{1,1} = \cdots = \gamma_{q,q} = \gamma \tag{2.34}
$$

then, the same LU decomposition can be used for all the stages, thus saving computation time.

In short form, the preceding equations which define a q-stage Rosenbrock's method for an autonomous system are given by

$$
(\mathbf{I} - h\gamma_{i,i}\mathbf{J}_k)\mathbf{k}_i = f\left(\mathbf{x}_k + h\sum_{j=1}^{i-1}\beta_{i,j}\mathbf{k}_j\right) + h\mathbf{J}_k\sum_{j=1}^{i-1}\gamma_{i,j}\mathbf{k}_j; \quad \mathbf{x}_{k+1} = \mathbf{x}_k + h\sum_{i=1}^{q}\omega_i\mathbf{k}_i \tag{2.35}
$$

If we now consider a *nonautonomous* system of equations (explicit dependence on t)

$$
\frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t} = \mathbf{f}(\mathbf{x}, t) \tag{2.36}
$$

then this system can be transformed into an autonomous form as

$$
\frac{\mathrm{d}\mathbf{x}}{\mathrm{d}\tau} = \mathbf{f}(\mathbf{x}, t); \qquad \frac{\mathrm{d}t}{\mathrm{d}\tau} = 1 \tag{2.37}
$$

and Eq. (2.35) lead to

$$
\begin{aligned}
(\mathbf{I} - h\gamma_{i,i}\mathbf{J}_k)\mathbf{k}_i = &f\left(\mathbf{x}_k + h\sum_{j=1}^{i-1}\beta_{i,j}\mathbf{k}_j, t_k + h\beta_i\right) \\
&+ h\mathbf{J}_k\sum_{j=1}^{i-1}\gamma_{i,j}\mathbf{k}_j + h\left.\frac{\partial\mathbf{f}}{\partial t}\right|_{t_k,\mathbf{x}_k}\gamma_i
\end{aligned} \tag{2.38}
$$

with $\beta_i = \sum_{j=1}^{i-1}\beta_{i,j}$ and $\gamma_i = \sum_{j=1}^{i}\gamma_{i,j}$.

For instance, a first-order accurate Rosenbrock's scheme for nonautonomous equations is given by

$$(\mathbf{I} - h\mathbf{J}_k)\mathbf{k}_1 = \mathbf{f}\left(\mathbf{x}_k, t_k\right) + h \left.\frac{\partial \mathbf{f}}{\partial t}\right|_{t_k, \mathbf{x}_k} \tag{2.39}$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + h\mathbf{k}_1 \tag{2.40}$$

A second-order accurate Rosenbrock's method developed for autonomous equations in [11], and called ROS2, is as follows:

$$(\mathbf{I} - \gamma h\mathbf{J}_k)\mathbf{k}_1 = \mathbf{f}\left(\mathbf{x}_k\right) \tag{2.41}$$

$$(\mathbf{I} - \gamma h\mathbf{J}_k)\mathbf{k}_2 = \mathbf{f}\left(\mathbf{x}_k + h\mathbf{k}_1\right) - 2\mathbf{k}_1 \tag{2.42}$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + 1.5h\mathbf{k}_1 + 0.5h\mathbf{k}_2 \tag{2.43}$$

with desirable stability properties for $\gamma \geq 0.25$.

In the following, we will focus attention on a modified Rossenbrock's method originally proposed in [12] and specifically designed for the solution of nonlinear parabolic problems, which will be of interest to us in the following chapters dedicated to the method of lines solutions of partial differential equations. As we have just seen, the main advantage of Rosenbrock's methods is to avoid the solution of nonlinear equations, which naturally arise when formulating an implicit method. In [12], the authors establish an efficient third-order Rosenbrock's solver for nonlinear parabolic PDE problems, which requires only three stages. In mathematical terms, the method described in [12] is stated in a transformed form, which is used in practice to avoid matrix-vector operations

$$\left(\frac{\mathbf{I}}{h\gamma} - \frac{\partial \mathbf{f}}{\partial \mathbf{x}}(\mathbf{x}_k, t_k)\right)\mathbf{X}_{k,i} = \mathbf{f}\left(\mathbf{x}_k + \sum_{j=1}^{i-1} a_{i,j}\mathbf{X}_{k,j}, t_k + h\alpha_i\right) + \sum_{j=1}^{i-1} \frac{c_{i,j}}{h}\mathbf{X}_{k,j} + hd_i \frac{\partial \mathbf{f}}{\partial t}(\mathbf{x}_k, t_k) \quad i = 1, 2, 3 \tag{2.44}$$

Two stepping formulas for a third- and a second-order methods, respectively, are used to estimate the truncation error, i.e., can be computed by taking the difference between the two following solutions

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \sum_{j=1}^{3} m_j \mathbf{X}_{k,j} \tag{2.45a}$$

Table 2.5 Parameters of the ROS23P algorithm

$\gamma = 0.5 + \sqrt{3}/6$	
$a_{21} = 1.267949192431123$	$c_{21} = -1.607695154586736$
$a_{31} = 1.267949192431123$	$c_{31} = -3.464101615137755$
$a_{32} = 0$	$c_{32} = -1.732050807567788$
$\alpha_1 = 0$	$d_1 = 0.7886751345948129$
$\alpha_2 = 1$	$d_2 = -0.2113248654051871$
$\alpha_3 = 1$	$d_3 = -1.077350269189626$
$m_1 = 2$	$\hat{m}_1 = 2.113248654051871$
$m_2 = 0.5773502691896258$	$\hat{m}_2 = 1$
$m_3 = 0.4226497308103742$	$\hat{m}_3 = 0.4226497308103742$

$$\hat{\mathbf{x}}_{k+1} = \mathbf{x}_k + \sum_{j=1}^{3} \hat{m}_j \mathbf{X}_{k,j} \tag{2.45b}$$

Note that the solutions at intermediate points, $\mathbf{X}_{k,j}$, are the same for both orders reflecting the embedding of the second-order method in the third-order method.

γ, $a_{i,j}$, $c_{i,j}$, α_i, d_i, m_i, $\hat{m}_i$ are parameters defined for a particular Rosenbrock's method. Particular values are listed in Table 2.5, which defines a method designated as *ROS23P*. *ROS* denotes *Rosenbrock*, 23 indicates a second-order method embedded in a third-order method in analogy with the fourth-order method embedded in a fifth-order method in *RKF45*, and *P* stands for *parabolic problems*. This solver will indeed be particularly useful for the time integration of ODE systems arising from the application of the method of lines to parabolic PDE problems. The selection of appropriate parameters, as in Table 2.5, confers desirable stability properties to the algorithm, which can therefore be applied to stiff ODEs. To reiterate, the system of Eq. (2.44) is *linear* in $\mathbf{X}_{k,j}$ which is a very favorable feature, i.e., a single application of a linear algebraic equation solver is all that is required to take the next step along the solution (from point k to $k + 1$). This is in contrast to implicit ODE methods which require the solution of *simultaneous nonlinear equations* that generally is a substantially more difficult calculation than solving linear equations (usually done by Newton's method, which must be iterated until convergence).

As in the case of *RKF45*, we program the algorithm in two steps: (1) a single step function analogous to `ssrkf45` and (2) a function that calls the single step routine to step along the solution and adjust the integration step that is analogous to `rkf45_solver`. We first list the single step routine, `ssros23p`

```
function [t,x,e] = ssros23p(odefunction,jacobian,...
                            time_derivative,t0,x0,h,gamma,a21,...
                            a31,a32,c21,c31,c32,d,alpha,m,mc)
%
% Function ssros3p computes an ODE solution by an implicit
% third–order Rosenbrock method for one step along the
```

```
% solution (by calls to 'odefunction' to define the ODE
% derivative vector, calls to 'jacobian' to define the
% Jacobian and calls to time_derivative if the problem is
% non autonomous).
%
% Argument list
%
% odefunction - string containing name of user-supplied problem
% jacobian - string containing name of user-supplied Jacobian
% time_derivative - sting containing name of user-supplied
%                   function time derivative
%
% t0 - initial value of independent variable
% x0 - initial condition vector
% h - integration step
% t - independent variable (scalar)
% x - solution vector after one rkf45 step
% e - estimate of truncation error of the solution vector
%
%   gamma,a21,a31,a32,c21,c31,c32,alpha,d,m,mc are the
%   method parameters

% Jacobian matrix at initial (base) point
[Jac] = feval(jacobian, t0, x0);

% Time derivative at initial (base) point
[Ft] = feval(time_derivative, t0, x0);

% Build coefficient matrix and perform L-U decomposition
CM = diag(1/(gamma*h)*ones(length(x0),1)) - Jac;
[L,U] = lu(CM);

% stage 1
xs = x0;
[xt] = feval(odefunction, t0+alpha(1)*h, xs);
rhs = xt + h*d(1)*Ft;
xk1 = U\(L\rhs);

% stage 2
xs = x0 + a21*xk1;
[xt] = feval(odefunction, t0+alpha(2)*h, xs);
rhs = xt + (c21/h)*xk1 + h*d(2)*Ft;
xk2 = U\(L\rhs);

% stage 3
xs = x0 + a31*xk1 + a32*xk2;
[xt] = feval(odefunction, t0+alpha(3)*h, xs);
rhs = xt + (c31/h)*xk1 + (c32/h)*xk2 + h*d(3)*Ft;
xk3 = U\(L\rhs);

% second-order step
x2 = x0 + mc(1)*xk1 + mc(2)*xk2 + mc(3)*xk3;

% third-order step
x3  = x0 + m(1)*xk1 + m(2)*xk2 + m(3)*xk3;

% error evaluation
t = t0 + h;
e = x3 - x2;
x = x3;
```

Function ssros23p Routine for a single step along the solution

We can note the following details about `ssros23p`:

1. The first statement defining function `ssros23p` has an argument list that includes the parameters (constants) used in the *ROS23P* algorithm as defined in Table 2.5.
2. After an initial set of comments explaining the operation of `ssros23p` and its arguments, the Jacobian matrix of the ODE system is first computed

```
% Jacobian matrix at initial (base) point
[Jac] = feval(jacobian,t0,x0);
```

   The user-supplied routine `jacobian` (to be discussed subsequently) is called to compute the Jacobian matrix in Eq. (2.44), $f_x(x_k, t_k)$, at the base point $t_k = t_0$, $x_k = x0$.
3. Then, the time derivative of the ODE function is computed

```
%...
%... Time derivative at initial (base) point
     [Ft] = feval(time_derivative, t0, x0);
```

   The user-supplied routine `time_derivative` is called to compute $f_t(x_k, t_k)$.
4. The LHS coefficient matrix of Eq. (2.44)

$$\left(\frac{I}{h\gamma} - f_x(\mathbf{x}_k, t_k)\right)$$

   is then constructed

```
% Build coefficient matrix and perform L-U decomposition
CM    = diag(1/(gamma*h)*ones(length(x0),1))-Jac;
[L,U] = lu(CM);
```

   Note that the main diagonal of the identity matrix is coded as `ones(length(x0),1)`. Then subtraction of the Jacobian matrix, `Jac`, results in a square coefficient matrix, `CM`, with dimensions equal to the length of the ODE dependent variable vector (`length(x0)`). After `CM` is constructed, it is factored (decomposed) into `L` and `U` lower and upper triangular factors using the MATLAB utility `lu`.

   Just to briefly review why this decomposition is advantageous, if the coefficient, A, of a linear algebraic system

$$Ax = b \tag{2.46a}$$

   is written in factored form, i.e., $A = LU$, Eq. (2.46a) can be written as

$$LUx = b \tag{2.46b}$$

Equation (2.46b) can be written as two algebraic equations

$$Ly = b \tag{2.47}$$

$$Ux = y \tag{2.48}$$

Note that Eq. (2.47) can be solved for y and Eq. (2.48) can then be solved for x (the solution of Eq. (2.46a)). If matrix A can be decomposed into a lower triangular matrix L and and upper triangular matrix U, then the solution of (2.47) can be easily done by elementary substitution starting from the first equation to the last one (forward substitution), whereas Eq. (2.48) can then be solved in the same way from the last to the first equation (backward substitution). This is a substantial simplification. Also, once the LU decomposition is performed, it can be used repeatedly for the solution of Eq. (2.46a) with different RHS vectors, b, as we shall observe in the coding of `ssros23p`. This reuse of the LU factorization is an important advantage since this factorization is the major part of the computational effort in the solution of linear algebraic equations.

5. We now step through the first of three stages, using $i = 1$ in Eq. (2.44)

```
% Stage 1
xs    = x0;
[xt] = feval(odefunction,t0+alpha(1)*h,xs);
rhs   = xt;
xk1   = U\(L\rhs);
```

Thus, the RHS of Eq. (2.44) is simply $xt = f(x_k, t_k)$ (from Eq. 1.12). Finally, the solution of Eq. (2.44) for X_{k1} is obtained by using the MATLAB operator twice, corresponding to the solution of Eqs. (2.47) and (2.48).

6. The second stage is then executed essentially in the same way as the first stage, but with $i = 2$ in Eq. (2.44) and using X_{k1} from the first stage

```
% Stage 2
xs    = x0 + a21*xk1;
[xt] = feval(odefunction,t0+alpha(2)*h,xs);
rhs   = xt + (c21/h)*xk1;
xk2   = U\(L\rhs);
```

Note in particular how the LU factors are used again (they do not have to be recomputed at each stage).

7. Finally, the third stage is executed essentially in the same way as the first and second stages, but with $i = 3$ in Eq. (2.44) and using X_{k1} from the first stage and X_{k2} from the second stage

```
% Stage 3
xs    = x0 + a31*xk1 + a32*xk2;
[xt] = feval(odefunction,t0+alpha(3)*h,xs);
rhs   = xt + (c31/h)*xk1 + (c32/h)*xk2;
xk3   = U\(L\rhs);
```

Again, the LU factors can be used as originally computed.

8. Equations (2.45a) and (2.45b) are then used to step the second- and third-order solutions from k to $k+1$

```
% Second-order step
x2 = x0 + mc(1)*xk1 + mc(2)*xk2 + mc(3)*xk3;
% Third-order step
x3 = x0 + m(1)*xk1 + m(2)*xk2 + m(3)*xk3;
```

9. The truncation error can be estimated as the difference between the second- and third-order solutions

```
% Estimated error and solution update
e = x3 - x2;
t = t0 + h;
x = x3;
```

The solution is taken to be the third-order result at the end of `ssros23p`.

In order to use function `ssros23p`, we require a calling function that will also adjust the integration step in accordance with user-specified tolerances and the estimated truncation error from `ssros23p`. A routine, analogous to `rkf45_solver`, is listed in `ros23p_solver`.

```
function [tout, xout, eout] = ros23p_solver(odefunction,...
                              jacobian,time_derivative,t0,tf,...
                              x0,hmin,nstepsmax,abstol,reltol,...
                              Dtplot)
% [tout, yout] = ros23p_solver('f','J','Ft',t0,tf,x0,hmin,...
%                               nstepsmax,abstol, reltol,Dtplot)
% Integrates a non-autonomous system of differential equations
% y'=f(t,x) from t0 to tf with initial conditions x0.
%
% Each row in solution array xout corresponds to a value
% returned in column vector tout.
% Each row in estimated error array eout corresponds to a
% value returned in column vector tout.
%
% ros23p_solver.m solves first-order differential equations
% using a variable-step implicit Rosenbrock method for a
% series of points along the solution by repeatedly calling
% function ssros23p for a single Rosenbrock step.
%
% The truncation error is estimated along the solution to
% adjust the integration step according to a specified error
% tolerance.
%
% Argument list
%
% f         - String containing name of user-supplied
%             problem description
%                    Call: xdot = fun(t,x) where f = 'fun'
%                    t      - independent variable (scalar)
%                    x      - Solution vector.
%                    xdot   - Returned derivative vector;
%                             xdot(i) = dx(i)/dt
%
% J         - String containing name of user-supplied Jacobian
```

```
%                    Call: Jac = fun(t,x) where J = 'fun'
%                    t      - independent variable (scalar)
%                    x      - Solution vector.
%                    Jac    - Returned Jacobian matrix;
%                             Jac(i,j) = df(i)/dx(j)
%
% Ft        - String containing nam of user-supplied
%             function time derivative
%                    Call: Ft = fun(t,x) where Ft = 'fun'
%                    t      - independent variable (scalar)
%                    x      - Solution vector.
%                    Ft     - Returned time derivative;
%                             Ft(i) = df(i)/dt
%
% t0        - Initial value of t
% tf        - Final value of t
% x0        - Initial value vector
% hmin      - minimum allowable time step
% nstepsmax - maximum number of steps
% abstol    - absolute error tolerance
% reltol    - relative error tolerance
% Dtplot    - Plot interval
%
% tout      - Returned integration points (column-vector).
% xout      - Returned solution, one solution row-vector per
%             tout-value.

% Initial integration step
h = 10*hmin;

% method parameters
gamma = 0.5+sqrt(3)/6;
a21 = 1.267949192431123;
a31 = 1.267949192431123;
a32 = 0.0;
c21 =-1.607695154586736;
c31 = -3.464101615137755;
c32 = -1.732050807567788;
d(1) = 7.886751345948129e-01;
d(2) = -2.113248654051871e-01;
d(3) = -1.077350269189626e+00;
alpha(1) = 0;
alpha(2) = 1.0;
alpha(3) = 1.0;
m(1) = 2.000000000000000e+00;
m(2) = 5.773502691896258e-01;
m(3) = 4.226497308103742e-01;
mc(1) = 2.113248654051871e+00;
mc(2) = 1.000000000000000e+00;
mc(3) = 4.226497308103742e-01;
% Start integration
t = t0;
tini = t0;
tout = t0;                          % initialize output value
xout = x0';                         % initialize output value
eout = zeros(size(xout));           % initialize output value
nsteps = 0;                         % initialize step counter
nplots = round((tf-t0)/Dtplot);  % number of outputs

% Initial integration step
h = 10*hmin;

% Step through nplots output points
for i = 1:nplots

    %   Final (output) value of the independent variable
```

```
    tplot = tini+i*Dtplot;
    % While independent variable is less than the final value,
    % continue the integration
    while t <= tplot*0.9999
        % If the next step along the solution will go past the
        % final value of the independent variable, set the step
        % to the remaining distance to the final value
        if t+h > tplot, h = tplot-t; end
        %  Single ros23p step
        [t,x,e] = ssros23p(odefunction,jacobian,...
            time_derivative,t0,x0,h,gamma,a21,a31,a32,...
            c21,c31,c32,d,alpha,m,mc);

        % Check if any of the ODEs have violated the error
        % criteria
        if max( abs(e) > (abs(x)*reltol + abstol) )
            % Error violation, so integration is not complete.
            % Reduce integration step because of error violation
            % and repeat integration from the base point. Set
            % logic variable for rejected integration step.
            h = h/2;

            % If the current step is less than the minimum
            % allowable step, set the step to the minimum
            % allowable value
            if h < hmin,  h = hmin; end

            % If there is no error violation, check if there is
            % enough "error margin" to increase the integration
            % step
        elseif max( abs(e) > (abs(x)*reltol + abstol)/8 )
            % The integration step cannot be increased, so leave
            % it unchanged and continue the integration from the
            % new base point.
            x0 = x; t0 = t;

            % There is no error violation and enough "security
            % margin"
        else

            % double integration step and continue integration
            % from new base point
            h = 2*h; x0 = x; t0 = t;
        end %if

        % Continue while and check total number of integration
        % steps taken
        nsteps=nsteps+1;
        if(nsteps > nstepsmax)
      fprintf('\n nstepsmax exceeded; integration terminated\n');
      break;
        end
    end %while

    % add latest result to the output arrays and continue for
    % loop
    tout = [tout ; t];
    xout = [xout ; x'];
    eout = [eout ; e'];
end % for
%
% End of ros23p_solver
```

Function ros23p_solver Routine for a variable step solution of an ODE system

ros23p_solver closely parallels rkf45_solver, so we point out just a few differences:

1. After a set of comments explaining the arguments and operation of ros23p_solver, the parameters for the *ROS23P* algorithm are set in accordance with the values defined in Table 2.5.
2. The variable step algorithm is then implemented as discussed for rkf45_solver, based on the estimated truncation error vector from a call to ssros23p

```
function [t,x,e] = ssros23p(odefunction,jacobian,...
                            time_derivative,t0,x0,h,gamma,...
                            a21,a31,a32,c21,c31,c32,d,...
                            alpha,m,mc)
```

In order to apply this algorithm to an example we need function to define the ODE model. If the logistic equation is chosen as an example, such function is the same as in logistic_ode. The exact solution to the logistic equation is again programmed in logistic_exact.

Since *ROS23P* requires the Jacobian matrix in ssros23p, we provide function logistic_jacobian for this purpose.

```
function [Jac] = logistic_jacobian(t , N)
% Set global variables
global a b K N0

% Jacobian of logistic equation Nt=(a-b*N)*N
Jac = a - 2*b*N;
```

Function logistic_jacobian Routine for the computation of the Jacobian matrix of logistic equation (2.1)

Note that, in this case, the logistic equation has a 1×1 Jacobian matrix (single element), which is just the derivative of the RHS of Eq. (2.1) with respect to N. ROS23P also requires a function to compute the time derivative of the ODE function, which is only useful for nonautonomous problems (so not in this case).

We now have all of the programming elements for the *ROS23P* solution of the logistic equation. Execution of the main program gives the numerical output of Table 2.6 (the plotted output is essentially identical to Fig. 2.2 and therefore is not repeated here).

A comparison of Tables 2.3 and 2.6 clearly indicates that *RKF45* in this case was much more effective in controlling the integration error than *ROS23P* (all of the integration parameters such as error tolerances were identical for the two integrators, i.e., the parameters of Table 2.2). However, *ROS23P* did meet the relative error tolerance, relerr = 1e-03. As explained previously, the large absolute error is due to the dominance of the total error by the relative error, which produces a typical

Table 2.6 Numerical output obtained with `ros23p_solver` for the logistic equation (2.1)

t	x(t)	xex(t)	abserr	relerr
0.0	1000.00	1000.00	0.00000	0.00000
0.5	1596.67	1596.92	-0.25079	-0.00016
1.0	2501.33	2503.22	-1.88884	-0.00075
1.5	3814.04	3817.18	-3.13425	-0.00082
2.0	5595.63	5600.09	-4.46446	-0.00080
2.5	7808.18	7813.68	-5.49428	-0.00070
3.0	10271.87	10277.73	-5.86125	-0.00057
3.5	12703.07	12708.50	-5.42835	-0.00043
4.0	14832.45	14836.83	-4.37216	-0.00029
4.5	16511.65	16514.31	-2.66122	-0.00016
5.0	17728.93	17730.17	-1.23181	-0.00007
⋮	⋮	⋮	⋮	⋮
15.0	19999.89	19999.88	0.00746	0.00000

term in the error test of `ros23p_solver` at `t = 1.5` of `3814.04(0.001) = 3.814` whereas the absolute error (in absolute terms) was smaller, i.e., `3.13425`. Thus, although *ROS23P* did not control the integration error as tightly as *RKF45*, its performance can still probably be considered acceptable for most applications, i.e., 1 part in 1,000. This poorer error control can be explained by the lower order of the error estimator of *ROS23P* relative to *RKF45*.

In other words, the error tolerances for *ROS23P* solver of `abserr = 1e-03`, `relerr = 1e-03` are rather loose. If they are tightened to `abserr = 1e-05`, `relerr = 1e-05` (i.e., 1 part in 10^5) the resulting output from `ros23p_solver` is

```
t       x(t)     xex(t)     abserr     relerr
0.0    1000.00    1000.00    0.00000    0.00000
0.5    1596.92    1596.92   -0.00623   -0.00000
1.0    2503.20    2503.22   -0.01681   -0.00001
1.5    3817.15    3817.18   -0.03137   -0.00001
2.0    5600.04    5600.09   -0.04699   -0.00001
2.5    7813.62    7813.68   -0.05887   -0.00001
3.0   10277.67   10277.73   -0.06277   -0.00001
3.5   12708.44   12708.50   -0.05726   -0.00000
4.0   14836.78   14836.83   -0.04438   -0.00000
4.5   16514.29   16514.31   -0.02130   -0.00000
5.0   17730.19   17730.17    0.02374    0.00000
.                               .
.                               .
.                               .
15.0  19999.89   19999.88    0.00639    0.00000
```

The enhanced performance of *ROS23P* using the tighter error tolerances is evident (*ROS23P* achieved the specified accuracy of 1 part in 10^5). This example illustrates the importance of error tolerance selection and the possible differences

in error control between different integration algorithms. In other words, some experimentation with the error tolerances may be required to establish the accuracy (reliability) of the computer solutions.

Since *ROS23P* has a decided advantage over *RKF45* for stiff problems (because of superior stability), we now illustrate this advantage with the 2×2 ODE system of Eq. (2.21) with again evaluating the numerical solution using the analytical solution of Eq. (2.23). The coding of the 2×2 problem to run under *ROS23P* is essentially the same as for *RKF45* and can also be found in the companion library. However, a routine for the Jacobian matrix is required by *ROS23P* (see function `jacobian_stiff_odes`).

```
function Jac = jacobian_stiff_odes(t,x)

% Set global variables
global a b

% Jacobian matrix
Jac = [-a   b;
        b  -a];
```

Function jacobian_stiff_odes Jacobian matrix of 2×2 ODE Eq. (2.21)

Here we are evaluating the Jacobian matrix $f_x(x_k, t_k)$ in Eq. (2.44) as required in `ssros23p`. For example, the first row, first element of this matrix is $\frac{\partial f_1}{\partial x_1} = -a$. Note that since Eq. (2.21) are *linear constant coefficient ODEs, their Jacobian matrix is a constant matrix*, and therefore function `jacobian_stiff_odes` would only have to be called once. However, since `ros23p_solver` and `ssros23p` are general purpose routines (they can be applied to nonlinear ODEs for which the Jacobian matrix is not constant, but rather is a function of the dependent variable vector), `jacobian_stiff_odes` will be called at each point along the solution of Eq. (2.21) through `ros23p_solver`.

We should note the following points concerning the Jacobian matrix:

1. If we are integrating an nth order ODE system (n first-order ODEs in n unknowns or a $n \times n$ ODE system), the *Jacobian matrix is of size* $n \times n$. This size increases very quickly with n. For example, if $n = 100$ (a modest ODE problem), the Jacobian matrix is of size $100 \times 100 = 10{,}000$.
2. In other words, we need to compute the $n \times n$ partial derivatives of the Jacobian matrix, and for large n (e.g., $n > 100$), this becomes difficult if not essentially impossible (not only because there are so many partial derivatives, but also, because the actual analytical differentiation may be difficult depending on the complexity of the derivative functions in the RHS of Eq. (1.12) that are to be differentiated).
3. Since analytical differentiation to produce the Jacobian matrix is impractical for large n, we generally have to resort to a numerical procedure for computing the required partial derivatives. Thus, a *numerical Jacobian* is typically used in

Table 2.7 Numerical output from `ssros23p` for the stiff problem (2.21) with $a = 500000.5$ and $b = 499999.5$

a = 500000.500			b = 499999.500	
t	x(t)	xex(t)	abserr	relerr
0.00	0.0000000	0.0000000	0.0000000	0.0000000
	2.0000000	2.0000000	0.0000000	0.0000000
0.10	0.9048563	0.9048374	0.0000189	0.0000208
	0.9048181	0.9048374	-0.0000193	-0.0000213
0.20	0.8187392	0.8187308	0.0000084	0.0000103
	0.8187187	0.8187308	-0.0000120	-0.0000147
0.30	0.7408210	0.7408182	0.0000028	0.0000038
	0.7408101	0.7408182	-0.0000082	-0.0000110
0.40	0.6703187	0.6703200	-0.0000013	-0.0000020
	0.6703128	0.6703200	-0.0000072	-0.0000108
0.50	0.6065277	0.6065307	-0.0000029	-0.0000049
	0.6065246	0.6065307	-0.0000061	-0.0000100
⋮	⋮	⋮	⋮	⋮
10.00	0.0000454	0.0000454	-0.0000000	-0.0007765
	0.0000454	0.0000454	-0.0000000	-0.0007765

the solution of stiff ODE systems. The calculation of a numerical Jacobian for Eq. (2.21) is subsequently considered.

The numerical output from these functions is listed in abbreviated form in Table 2.7. As in the case of Table 2.4, two rows are printed at each time instant. The first row corresponds with the output for the first state variable (x_1) while the second row corresponds with the second state variable (x_2). This solution was computed with good accuracy and modest computation effort (the number of calls to IVP solver was `nsteps = 14`). Note how the two solutions, $x_1(t)$, $x_2(t)$, merged almost immediately and were almost identical by $t = 0.1$ (due to the large eigenvalue $\lambda_2 = -10^6$ in Eq. (2.24) so that the exponential $e^{-\lambda_2 t}$ decayed to insignificance almost immediately). This example clearly indicates the advantage of a stiff integrator (*RKF45* could not handle this problem with reasonable computational effort since it would take an extremely small integration step because of the stiffness). Clearly the solution of the simultaneous equations of Eq. (2.44) was worth the effort to maintain stability with an acceptable integration step (the integration step could be monitored by putting it in an output statement in `ros23p_solver`). Generally, this example illustrates the advantage of an implicit (stiff) integrator (so that the additional computation of solving simultaneous linear algebraic equations is worthwhile).

There is one detail that should be mentioned concerning the computation of the solution in Table 2.7. Initially `ros23p_solver` and `ssros23p` failed to compute a solution. Some investigation, primarily by putting output statements in

`ros23p_solver`, indicated that the problem was the selection of an initial integration interval in `ros23p_solver` according to the statement

```
% Initial integration step
h = Dtplot/10;
```

In other words, since `Dtplot = 0.1` (see the output interval of Table 2.7), the initial integration step is 0.01. Recall that `ros23p_solver` is trying to compute a solution according to Eq. (2.24) in which an exponential decays according to $e^{-10^6 t}$. If the initial step in the numerical integration is $h = 0.01$, the exponential would be $e^{-10^6(0.01)} = e^{-10^4}$ and it therefore has decayed to insignificance. The automatic adjustment of the integration step in `ros23p_solver` would have to reduce the integration step from 0.01 to approximately 10^{-7} so that the exponential is $e^{-10^6(10^{-7})} = e^{-0.1}$ and it apparently was unable to do this. By changing the initial integration programming to

```
% Initial integration step
h = Dtplot/1.0e+5;
```

the numerical integration proceeded without difficulty and produced the numerical solution of Table 2.7. This discussion illustrates that some experimentation with the initial integration step may be required, particularly for stiff problems. An alternative would be to use an available algorithm for the initial integration step, but this would increase the complexity of `ros23p_solver`. We therefore opted to use a manual adjustment of the initial integration interval to successfully start the numerical integration.

As indicated previously, the use of an analytical Jacobian with a stiff integrator is not practical for large ODE problems (i.e., $n > 100$), but rather, a numerical Jacobian should be used. To illustrate this approach, we consider the use of a numerical Jacobian for the solution of Eq. (2.21). For example, we can use the *finite difference approximations*.

$$f_x = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix} \approx \begin{bmatrix} \frac{f_1(x_1+\Delta x_1, x_2)-f_1(x_1,x_2)}{\Delta x_1} & \frac{f_1(x_1, x_2+\Delta x_2)-f_1(x_1,x_2)}{\Delta x_2} \\ \frac{f_2(x_1+\Delta x_1, x_2)-f_2(x_1,x_2)}{\Delta x_1} & \frac{f_2(x_1, x_2+\Delta x_2)-f_2(x_1,x_2)}{\Delta x_2} \end{bmatrix} \tag{2.49}$$

Application of Eq. (2.49) to Eq. (2.21) gives

$$f_x \approx \begin{bmatrix} \frac{-a(x_1+\Delta x_1)+bx_2-(ax_1+bx_2)}{\Delta x_1} & \frac{-ax_1+b(x_2+\Delta x_2)-(-ax_1+bx_2)}{\Delta x_2} \\ \frac{f_2(x_1+\Delta x_1, x_2)-f_2(x_1,x_2)}{\Delta x_1} & \frac{f_2(x_1, x_2+\Delta x_2)-f_2(x_1,x_2)}{\Delta x_2} \end{bmatrix} = \begin{bmatrix} -a & b \\ b & -a \end{bmatrix}$$

Thus, Eq. (2.49) gives the exact Jacobian (see `jacobian_stiff_odes`) because the finite differences are exact for linear functions (i.e., the linear functions of Eq. (2.21)). However, generally this will not be the case (for nonlinear functions) and

therefore the finite difference approximation of the Jacobian matrix will introduce errors in the numerical ODE solution. The challenge then is to compute the numerical Jacobian with sufficient accuracy to produce a numerical solution of acceptable accuracy. Generally, this has to do with the selection of the increments Δx_1, Δx_2. But the principal advantage in using finite differences to avoid analytical differentiation is generally well worth the additional effort of computing a sufficiently accurate numerical Jacobian. To illustrate the programming of a numerical Jacobian, we use routine `jacobian_stiff_odes_fd` instead of `jacobian_stiff_odes`.

```
function [Jac] = jacobian_stiff_odes_fd(t,x)
% Function jacobian_stiff_odes_fd computes the Jacobian matrix
% by finite differences

% Set global variables
global a b

% Jacobian of the 2 x 2 ODE system
% Derivative vector at base point
xb  = x;
xt0 = stiff_odes(t,x);

% Derivative vector with x1 incremented
dx1  = x(1)*0.001 + 0.001;
x(1) = x(1) + dx1;
x(2) = xb(2);
xt1  = stiff_odes(t,x);

% Derivative vector with x2 incremented
x(1) = xb(1);
dx2  = x(2)*0.001 + 0.001;
x(2) = x(2) + dx2;
xt2  = stiff_odes(t,x);

% Jacobian matrix (computed by finite differences in place of
% Jac = [-a b; b -a]);
Jac(1,1) = (xt1(1)-xt0(1))/dx1;
Jac(1,2) = (xt2(1)-xt0(1))/dx2;
Jac(2,1) = (xt1(2)-xt0(2))/dx1;
Jac(2,2) = (xt2(2)-xt0(2))/dx2;
```

Function jacobian_stiff_odes_fd Routine for the computation of the Jacobian matrix of Eq. (2.21) using the finite differences of Eq. (2.49)

We can note the following points about this routine:

1. While the Jacobian routine `jacobian_stiff_odes_fd` appears to be considerably more complicated than the routine `jacobian_stiff_odes`, it has one important advantage: *analytical differentiation is not required*. Rather, the finite difference approximation of the Jacobian partial derivatives requires only calls to the ODE routine, `stiff_odes` (which, of course, is already available for the problem ODE system).
2. To briefly explain the finite differences as expressed by Eq. (2.49), we first need the derivative functions at the base (current) point along the solution (to give

the derivative functions $f_1(x_1, x_2), f_2(x_1, x_2)$ in Eq. (2.49)). These derivatives are computed by the first call to `stiff_odes`

3. Then we need the derivatives with x_1 incremented by Δx_1, that is $f_1(x_1 + \Delta x_1, x_2)$, $f_2(x_1 + \Delta x_1, x_2)$ which are computed by

```
% Derivative vector with x1 incremented
dx1  = x(1)*0.001 + 0.001;
x(1) = x(1) + dx1;
x(2) = xb(2);
xt1  = stiff_odes(t,x);
```

Note that in computing the increment `dx1` we use $0.001x_1 + 0.001$; the second 0.001 is used in case $x_1 = 0$ (which would result in no increment) as it does at the initial condition $x_1(0) = 0$.

4. Next, the derivatives with x_2 incremented by Δx_2 are computed

```
% Derivative vector with x2 incremented
x(1) = xb(1);
dx2  = x(2)*0.001 + 0.001;
x(2) = x(2) + dx2;
xt2  = stiff_odes(t,x);
```

5. Then the four partial derivatives of the Jacobian matrix are computed. For example, $\frac{\partial f_1}{\partial x_1} \approx \frac{f_1(x_1+\Delta x_1, x_2) - f_1(x_1, x_2)}{\Delta x_1}$ is computed as

```
% Jacobian matrix (computed by finite differences in
% place of Jac = [-a b; b -a]);
Jac(1,1) = (xt1(1)-xt0(1))/dx1;
```

The numerical solution with `jacobian_stiff_odes` (analytical Jacobian) replaced with `jacobian_stiff_odes_fd` (numerical Jacobian) gave the same solution as listed in Table 2.7. This is to be expected since the finite difference approximations are exact for the linear ODEs Eq. (2.21).

To summarize this discussion of *ROS23P*, we have found that this algorithm:

1. Has a reliable error estimate (the difference between the second- and third-order solutions), but not as accurate as *RKF45* (the difference between the fourth- and fifth-order solutions).
2. Requires only the solution of linear algebraic equations at each step along the solution (thus the name LIRK where "LI" denotes *linearly implicit*). We should also note that the accuracy of the numerical solution is directly dependent on the accuracy of the Jacobian matrix. This is in contrast with implicit ODE integrators that require the solution of nonlinear equations, but the Jacobian can often be approximate or inexact since all that is required is the convergence of the nonlinear equation solutions, usually by Newton's method or some variant, which can converge with an inexact Jacobian.
3. Has excellent stability properties (which are discussed in detail in [12]).

Table 2.8 MATLAB solvers for differential equations

Initial value problem solvers for ODEs (*if unsure about stiffness, try* `ode45` *first, then* `ode15s`)	
`ode45`	Solve nonstiff differential equations, medium-order method
`ode23`	Solve nonstiff differential equations, low-order method
`ode113`	Solve nonstiff differential equations, variable-order method
`ode23t`	Solve moderately stiff ODEs and DAEs index 1, trapezoidal rule
`ode15s`	Solve stiff ODEs and DAEs index 1, variable-order method
`ode23s`	Solve stiff differential equations, low-order method
`ode23tb`	Solve stiff differential equations, low-order method
Initial value problem solvers for fully implicit ODEs/DAEs $F(t, y, y') = 0$	
`decic`	Compute consistent intial conditions
`ode15i`	Solve implicit ODEs or DAEs i ndex 1
Initial value problem solver for delay differential equations (*DDEs*)	
`dde23`	Solve delay differential equations (DDEs) with constant delays
Boundary value problem solver for ODEs	
`bvp4c`	Solve two-point boundary value problems for ODEs by collocation
1D Partial differential equation solver	
`pdepe`	Solve initial-boundary value problems for parabolic-elliptic PDE

Thus, we now have a choice of high accuracy nonstiff (*RKF45*) and stiff (*ROS23P*) algorithms that are implemented in library routines for solution of the general $n \times n$ initial value ODE problem. The ambition of these solvers is not to compete with the high-quality integrators that are included in the MATLAB ODE Suite or within SCILAB or OCTAVE, but they are easy to use and to understand and will also be easily translated into various environments. Before testing them in additional applications, we introduce briefly the MATLAB ODE Suite.

2.4 MATLAB ODE Suite

We have previously discussed the use of advanced integrators, which are part of the MATLAB ODE suite or library. Examples include the use of `ode45` and `ode15s` in `Main_bacteria`, and `ode15s` in `Main_two_tanks`. The MATLAB ODE integrators have options beyond the basic integrators; furthermore, there is an extensive selection of integrators as given in Table 2.8. We cannot go into all of the features available because of limited space. The code for these integrators is relatively long and complex, so they are typically used without modification. Additional details about the MATLAB integrators are available in [1].

The application examples described in the next section are used to compare some of the time integrators developed in the previous sections, as well as several integrators from the MATLAB ODE Suite.

2.5 Some Additional ODE Applications

The ODE applications considered previously in this chapter were quite modest in complexity and could therefore be solved analytically or numerically; comparison of these two types of solutions was used to establish the validity of the numerical algorithms and associated codes. Now that the numerical integrators have been developed and tested, we consider some ODE applications that are sufficiently complex to preclude analytical solution; thus, we have only the numerical integrators as a viable approach to solutions, which is the usual case in realistic applications, i.e., numerical methods can be applied when analytical methods are not tractable.

2.5.1 Spruce Budworm Dynamics

The first application we consider is an ecological model of the interaction of spruce budworm populations with forest foliage [13]. The full model is a 3×3 system of logistic-type ODEs. However, before considering the full model, we consider a simplified version that has only one ODE, but which exhibits interesting dynamics. The idea of the simplified model is to consider that the slow variables (associated with foliage quantity and quality, $S(t)$) are held fixed, and to analyze the long-term behavior of the fast variable, i.e., the budworm density, $B(t)$.

The budworm's growth follows a logistic equation

$$\frac{\mathrm{d}B}{\mathrm{d}t} = r_B B\left(1 - \frac{B}{k_B}\right) - g \tag{2.50}$$

where the carrying capacity $k_B = kS$ is proportional to the amount of foliage available, i.e., proportional to S.

The effect of predation (consumption of budworms by birds) is represented by:

$$g = \frac{\beta B^2}{\alpha^2 + B^2} \tag{2.51}$$

which has the following properties:

- The consumption of prey (budworms) by individual predators (birds) is limited by saturation to the level β; note that $g \to \beta$ for large B.
- There is a decrease in the effectiveness of predation at low prey density (with the limit $g \to 0, B \to 0$, i.e., the birds have a variety of alternative foods).
- α determines the scale of budworm densities at which saturation takes place, i.e., α^2 relative to B^2.

Clearly g from Eq. (2.51) introduces a strong nonlinearity in ODE (2.50) which is a principal reason for using numerical integration in the solution of Eq. (2.50).

We can now use Eq. (2.50) to plot $\frac{\mathrm{d}B}{\mathrm{d}t}$ versus B as in Fig. 2.5, termed a *phase plane plot*. Note that there are three equilibrium points for which $\frac{\mathrm{d}B}{\mathrm{d}t} = 0$ (the fourth point

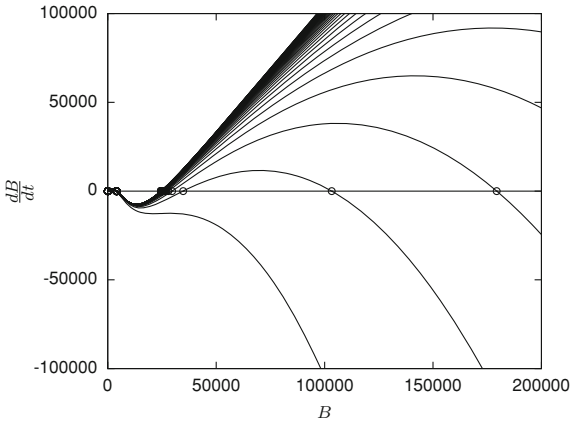

Fig. 2.5 $\frac{\mathrm{d}B}{\mathrm{d}t}$ as a function of B from Eq. (2.50), for increasing value of S (S evolves from 200 to 10,000 by steps of 200)

at the origin $B = 0$, $t = 0$ is not significant). These equilibrium points are the roots of Eq. (2.50) (combined with Eq. (2.51)) with $\frac{\mathrm{d}B}{\mathrm{d}t} = 0$ i.e.,

$$0 = r_B B\left(1 - \frac{B}{K_B}\right) - \frac{\beta B^2}{\alpha + B^2} \tag{2.52}$$

The middle equilibrium point (at approximately $B = 1.07 \times 10^5$) is not stable while the two outlying equilibrium points are stable. Figure 2.5 is parametrized with increasing values of S (from 200 to 10,000, by steps of 200) corresponding to increasing values of the carrying capacity $K_B = kS$.

Qualitatively, the evolution of the system can be represented as in Fig. 2.6. Consider the situation where there are three equilibrium points as in Fig. 2.5. In this case, the intermediate point is unstable, whereas the upper and lower equilibrium points are stable. Assume that the system is in a lower equilibrium point. As the forest foliage, $S(t)$, slowly increases, the system moves along the heavy lower equilibrium line. This happens because the budworm density, $B(t)$ is low (endemic population), and the forest can grow under this good condition (low $B(t)$). At the end of this slow process, the system moves quickly along a vertical arrow to reach the upper equilibrium branch, which corresponds to a much higher budworm density (outbreak population). Now, old trees are more susceptible to defoliation and die. As a consequence, $S(t)$ slowly decreases, and the system slowly moves along the heavy upper equilibrium line. This continues up to a certain stage, where the system jumps along a vertical line (budworm population collapses), and goes back to a status characterized by a low budworm density.

In summary, slow processes are depicted by the heavy equilibrium lines, whereas fast processes are represented by vertical arrows in Fig. 2.6. The interacting spruce budworm population and forest follow limit cycles characterized by two periods of

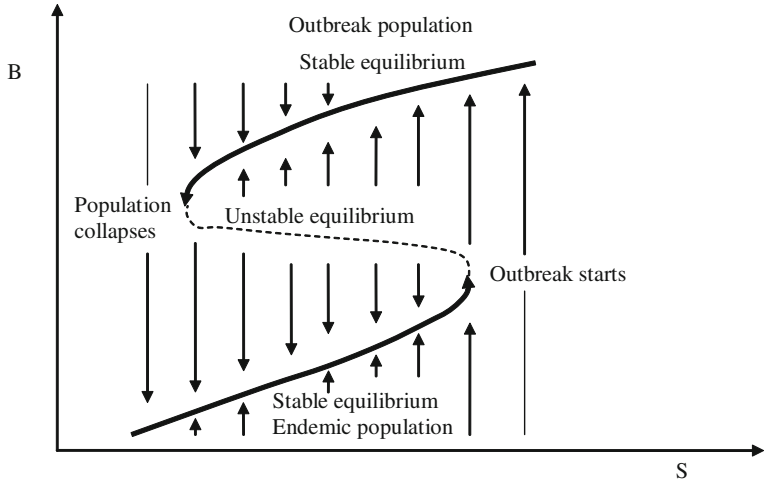

Fig. 2.6 Qualitative evolution of the budworm-forest system

slow change and two periods of fast change. An analysis of the system must include the fast processes along with the slow processes. Even though the system dynamics is dictated by the slow processes for most of the time (i.e., when the system moves slowly along heavy equilibrium lines), fast transitions explain budworm population outbreaks and collapses, which would be completely unforeseen if the fast dynamics is neglected.

We now consider the full model in which the carrying capacity is given by

$$K_B = \frac{kSE^2}{E^2 + T_E^2} \tag{2.53}$$

i.e., it is again proportional to the amount of foliage available, $S(t)$, but also depends on the physiological condition (energy) of the trees, $E(t)$; K_B declines sharply when E falls below a threshold T_E.

The effect of predation is still represented by Eq. (2.51), but, in addition, the half-saturation density α is proportional to the branch surface area S, i.e., $\alpha = aS$.

The total surface area of the branches in a stand then follows the ODE

$$\frac{\mathrm{d}S}{\mathrm{d}t} = r_S S \left(1 - \frac{S}{K_S}\frac{K_E}{E}\right) \tag{2.54}$$

that allows S to approach its upper limit K_S. An additional factor $\frac{K_E}{E}$ is inserted into the equation because S inevitably decreases under stress conditions (death of branches or even whole trees).

Table 2.9 Spruce budworm versus forest—parameter values

Parameter	Value	Units
r_B	1.52	year^{-1}
r_S	0.095	year^{-1}
r_E	0.92	year^{-1}
k	355	larvae/branch
a	1.11	larvae/branch
β	43,200	larvae/acre/year
K_S	25,440	branch/acre
K_E	1	–

The energy reserve also satisfies an equation of the logistic type

$$\frac{\mathrm{d}E}{\mathrm{d}t} = r_E E\left(1 - \frac{K_E}{E}\right) - P\frac{B}{S} \tag{2.55}$$

where the second term on the RHS describes the stress exerted on the trees by the budworm's consumption of foliage. In this expression $\frac{B}{S}$ represents the number of budworms per branch. The proportionality factor P is given by

$$P = \frac{pE^2}{E^2 + T_E^2} \tag{2.56}$$

as the stress on the trees is related to the amount of foliage consumed (P declines sharply when E falls below a threshold T_E).

The initial conditions (ICs) are taken as:

$$B(t=0) = 10; \qquad S(t=0) = 7,000; \qquad E(t=0) = 1 \tag{2.57}$$

The model parameters are given in Table 2.9, [13]. The 3×3 ODE model—Eqs. (2.50)–(2.56)—are solved by the code in function `spruce_budworm_odes`.

```
function xt = spruce_budworm_odes(t,x)

% Set global variables
global rb k beta a rs Ks re Ke p Te

% Transfer dependent variables
B = x(1);
S = x(2);
E = x(3);

% Model Parameters
Kb    = k*S*E^2/(E^2+Te^2);
alpha = a*S;
g     = beta*B^2/(alpha^2+B^2);
P     = p*E^2/(Te^2+E^2);
```

```
% Temporal derivatives
Bt  = rb*B*(1-B/Kb) - g;
St  = rs*S*(1-(S/Ks)*(Ke/E));
Et  = re*E*(1-E/Ke) - P*B/S;

% Transfer temporal derivatives
xt = [Bt St Et]';
```

Function spruce_budworm_odes Function for the solution of Eqs. (2.50)–(2.56) and associated algebraic equations

We can note the following details about this function:

1. After defining the global variables which are shared with the main program to be discussed next, the dependent variable vector received from the integration function, `x`, is transferred to problem oriente variables to facilitate programming

```
% Global variables
global rb k beta a rs Ks re Ke p Te

% Transfer dependent variables
B = x(1);
S = x(2);
E = x(3);
```

2. The problem algebraic variables are computed from the dependent variable vector $(B, S, E)^T$

```
% Temporal derivatives

Kb    = k*S*E^2/(E^2+Te^2); alpha = a*S; g     =
beta*B^2/(alpha^2+B^2); P     = p*E^2/(Te^2+E^2);
```

   Note the importance of ensuring that all variables and parameters on the RHS of these equations are set numerically before the calculation of the RHS variables.
3. The ODEs, Eqs. (2.50), (2.54), (2.55) are then programmed and the resulting temporal derivatives are transposed to a column vector as required by the integrator

```
% Temporal derivatives
Bt  = rb*B*(1-B/Kb) - g;
St  = rs*S*(1-(S/Ks)*(Ke/E));
Et  = re*E*(1-E/Ke) - P*B/S;
%
% Transfer temporal derivatives
xt = [Bt St Et]';
```

The main program that calls the ODE function `spruce_budworm_odes` is shown in `Budworm_main`

```
close all
clear all
```

```
% start a stopwatch timer
tic

% set global variables
global rb k beta a rs Ks re Ke p Te

% model parameters
rb   = 1.52;
k    = 355;
beta = 43200;
a    = 1.11;
rs   = 0.095;
Ks   = 25440;
re   = 0.92;
Ke   = 1.0;
p    = 0.00195;
Te   = 0.03;

% initial conditions
t0 = 0;
tf = 200;
B  = 10;
S  = 7000;
E  = 1;
x  = [B S E]';

% call to ODE solver (comment/decomment one of the methods
% to select a solver)

% method = 'Euler'
% method = 'rkf45'
% method = 'ode45'
 method = 'ode15s'
%
switch method
    % Euler
    case('Euler')
        Dt = 0.01;
        Dtplot = 0.5;
        [tout, yout] = euler_solver(@spruce_budworm_odes,...
                                    t0, tf, x, Dt, Dtplot);
    % rkf45
    case('rkf45')
        hmin = 1e-3;
        nstepsmax = 1000;
        abstol = 1e-3;
        reltol = 1e-3;
        Dtplot = 0.5;
        [tout,yout,eout] = rkf45_solver(@spruce_budworm_odes,...
                           t0,tf,x,hmin,nstepsmax,abstol,...
                           reltol,Dtplot);
        figure(5)
        plot(tout,eout)
        xlabel('t');
        ylabel('truncation error');
    % ode45
    case('ode45')
        options = odeset('RelTol',1e-6,'AbsTol',1e-6);
```

```
        t=[t0:0.5:tf];
        [tout, yout] = ode45(@spruce_budworm_odes,t,x,...
                             options);
    % ode15s
    case('ode15s')
        options = odeset('RelTol',1e-6,'AbsTol',1e-6);
        t=[t0:0.5:tf];
        [tout, yout] = ode15s(@spruce_budworm_odes,t,x,...
                              options);

end
% plot results
figure(1)
subplot(3,1,1)
plot(tout,yout(:,1),'k');
%     xlabel('t [years]');
ylabel('B(t)','FontName','Helvetica','FontSize',12);
%     title('Budworm density');
subplot(3,1,2)
plot(tout,yout(:,2),'k');
%     xlabel('t [years]');
ylabel('S(t)','FontName','Helvetica','FontSize',12);
%     title('Branch density')
subplot(3,1,3)
plot(tout,yout(:,3),'k');
xlabel('t [years]','FontName','Helvetica','FontSize',12);
ylabel('E(t)','FontName','Helvetica','FontSize',12);
set(gca,'FontName','Helvetica','FontSize',12);
%     title('Energy');
figure(2)
plot3(yout(:,1),yout(:,2),yout(:,3),'k')
xlabel('B(t)','FontName','Helvetica','FontSize',12);
ylabel('S(t)','FontName','Helvetica','FontSize',12);
zlabel('E(t)','FontName','Helvetica','FontSize',12);
grid
title('3D phase plane plot','FontName','Helvetica',...
      'FontSize',12);
set(gca,'FontName','Helvetica','FontSize',12);

% read the stopwatch timer
tcpu=toc;
```

Script Budworm_main Main program for the solution of Eqs. (2.50)–(2.56)

We can note the following points about this main program:

1. First, variables are defined as global so they can be shared with the ODE routine
2. The model parameters are then defined numerically
3. The time scale and the initial conditions of Eq. (2.57) are defined
4. The parameters of the integrator, e.g., error tolerances, are set and an integrator is selected among four possible choices, e.g., `euler_solver`, `rkf45_solver` (among the basic integrators introduced earlier in this chapter) or `ode45` and `ode15s` (among the integrators available in the MATLAB ODE suite). Table 2.10 shows a comparison of the performance of these and other IVP solvers applied to

Table 2.10 Performance of different IVP solvers applied to the spruce budworm problem

	Euler	Heun	rkf45	ros23p	ode45	ode15s Adams	BDF
N. steps	20,000	20,000	1,122	20,520	837	2,109	2,117
CPU time	2.5	4.9	1.0	4.24	1.1	2.2	2.2

Computational times are normalized with respect to the time spent by the most efficient solver, in this example, rkf45

Eqs. (2.50)–(2.56). Absolute and relative tolerances were fixed to 10^{-6}. For this particular example, the most efficient solver is rkf45 while Heun's method requires the highest computational cost. These results show that the system of ODEs is not stiff (rkf45 is an explicit time integrator) and that time step size adaptation is an important mechanism to ensure specified tolerances and to improve computational efficiency (we have used a conservatively small step size with fixed step integrators such as Euler's and Heun's method).

5. After calculating the solution, a series of plots displays the numerical solution.

The plotted solutions correspond with Fig. 2.7, for the time evolution of the three states, and Fig. 2.8, for the 3D phase plane. The oscillatory nature of the solution resulting from the combination of slow and fast dynamics depicted in Fig. 2.7 is clear. Also, the number of nonlinear ODEs (three) and associated algebraic equations demonstrates the utility of the numerical solution; in other words, analytical solution of this model is precluded because of its size and complexity.

2.5.2 Liming to Remediate Acid Rain

As a final ODE example application, we consider the modeling of an ecological system described by a 3×3 system of nonlinear ODEs. Specifically, the effect of acid rain on the fish population of a lake and the effect of remedial liming is investigated. This model is described in [14].

The fish population $N(t)$ is growing logistically, i.e.,

$$\frac{\mathrm{d}N}{\mathrm{d}t} = r(C)N - \frac{r_0 N^2}{K(C)} - H, \quad N(0) = N_0 > 0 \tag{2.58}$$

where $r(C)$ is the specific growth rate, which depends on the acid concentration $C(t)$ in the following way:

$$r(C) = \begin{cases} r_0 & \text{if} \quad C < C_{\text{lim}} \\ r_0 - \alpha(C - C_{\text{lim}}) & \text{if} \quad C_{\text{lim}} < C < C_{\text{death}} \\ 0 & \text{if} \quad C_{\text{death}} < C < Q/\delta \end{cases}, \tag{2.59}$$

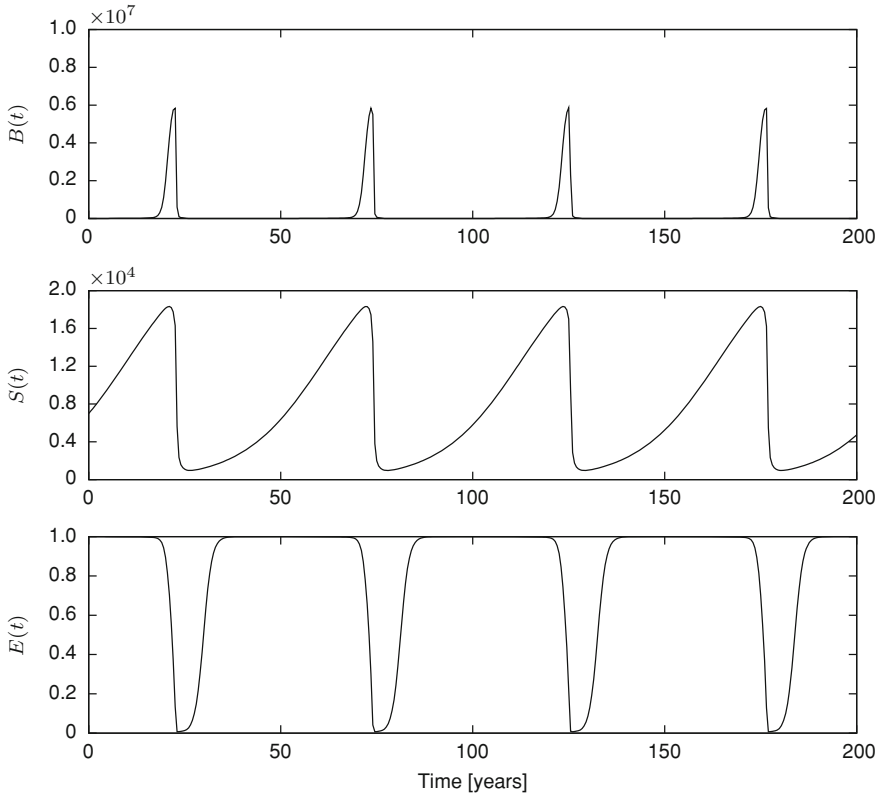

Fig. 2.7 Evolution of the three state variables ($B(t)$, $S(t)$ and $E(t)$) from the problem described by Eqs. (2.50), (2.54) and (2.55)

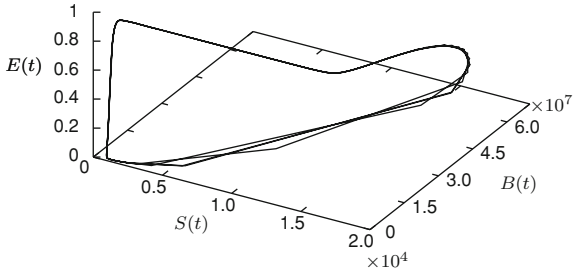

Fig. 2.8 Composite plot of the dependent variable vector from Eqs. (2.50), (2.54) and (2.55)

$K(C)$ is the carrying capacity (i.e., the maximum population density that the ecosystem can support), which also depends on $C(t)$

$$K(C) = \begin{cases} K_0 & \text{if} \quad C < C_{\text{lim}} \\ K_0 - \beta(C - C_{\text{lime}}) & \text{if} \quad C_{\text{lim}} < C < C_{\text{death}} \\ K_{\text{lim}} & \text{if} \quad C_{\text{death}} < C < Q/\delta \end{cases}, \tag{2.60}$$

Table 2.11 Table of parameter values for Eqs. (2.58)–(2.62)

$r0 = 0.02$	$Q = 2$
$C_{\text{lim}} = 50$	$\delta = 0.002$
$\alpha = 0.0001$	$\delta_0 = 0.005$
$\beta = 0.05$	$\eta = 0.04$
$K_0 = 100{,}000$	$\eta_0 = 0.004$
$K_{\text{lim}} = 100$	$H = 100$

and H is the harvesting rate. In Eqs. (2.59)–(2.60), C_{lim} denotes the critical value of the acid concentration (between 0 and C_{lim}, acid is harmless to the fish population) and $C_{\text{death}} = (r_0 - \alpha C_{\text{lime}})/\alpha$ is the concentration above with the fish population completely stops growing.

We suggest a careful study of the RHS of Eq. (2.58) and the switching functions of Eqs. (2.59)–(2.60) since these functions reflect several features of the model.

The acid concentration $C(t)$ is described by the ODE

$$\frac{\mathrm{d}C}{\mathrm{d}t} = Q - \delta C - \delta_0 E \tag{2.61}$$

where Q is the inlet acid flow rate (due to acid rain), δ is the natural depletion rate, while δ_0 is the depletion rate due to liming. E is the liming effort applied to maintain the lake at a permissible acid concentration C_{lim}, and is given by the ODE

$$\frac{\mathrm{d}E}{\mathrm{d}t} = \eta(C - C_{\text{lim}}) - \eta_0 E \tag{2.62}$$

where η represents the control action and η_0 is the natural depletion rate of E.

Again, we suggest a careful analysis of the RHS functions of Eqs. (2.61) and (2.62). The complexity of the model is clearly evident from Eqs. (2.58)–(2.62), e.g., the nonlinear switching functions of Eqs. (2.59) and (2.60). Thus, although some analytical analysis is possible as we demonstrate in the subsequent discussion, a numerical solution of Eq. (2.24) is the best approach to gain an overall understanding of the characteristics of the model. In particular, we will compute the state space vector (the solution of Eqs. (2.58), (2.61) and (2.62)), $N(t)$, $C(t)$, $E(t)$ by numerical ODE integration.

The initial conditions for Eqs. (2.58), (2.61) and (2.62) are taken as

$$N(t = 0) = 72{,}500; \quad C(t = 0) = 80; \quad E(t = 0) = 190 \tag{2.63}$$

and the parameter values are given in Table 2.11.

If harvesting is below a certain threshold value, i.e., $H < \frac{K(C^*)\{r(C^*)\}^2}{4r_0}$, there exist two equilibrium points $P_i(N^*, C^*, E^*)$ with $i = 1, 2$ in the state space. The notation * indicates that the state is at equilibrium.

$$E^* = \frac{\eta\left(\frac{Q}{\delta} - C_{\text{lim}}\right)}{\eta_0 + \eta\frac{\delta_0}{\delta}} \tag{2.64}$$

$$C^* = \frac{Q - \delta_0 E^*}{\delta} \tag{2.65}$$

$$N_1^* = \frac{K(C^*)r(C^*)}{2r_0}\left(1 - \sqrt{1 - \frac{4r_0 H}{K(C^*)\{r(C^*)\}^2}}\right) \tag{2.66}$$

$$N_2^* = \frac{K(C^*)r(C^*)}{2r_0}\left(1 + \sqrt{1 - \frac{4r_0 H}{K(C^*)\{r(C^*)\}^2}}\right) \tag{2.67}$$

The equilibrium point P_1 corresponding to N_1^* of Eq. (2.66) is unstable (two eigenvalues of the Jacobian matrix have negative real parts, but the third one is real positive, so that P_1 is a saddle point) whereas P_2 corresponding to N_2^* of Eq. (2.67) is locally asymptotically stable (the three eigenvalues of the Jacobian matrix have negative real parts). In fact, when C and E tends to their steady-state values C^* and E^*, it is possible to rewrite the ODE for N, Eq. (2.58), as follows

$$\frac{\mathrm{d}N}{\mathrm{d}t} = -\frac{r_0}{K(C^*)}(N - N_1^*)(N - N_2^*) \tag{2.68}$$

Since $N_2^* > N_1^*$, the RHS of Eq. (2.68) expression shows that

$$\frac{\mathrm{d}N}{\mathrm{d}t} < 0 \quad \text{if} \quad N < N_1^* \quad \text{or} \quad N > N_2^* \tag{2.69}$$

$$\frac{\mathrm{d}N}{\mathrm{d}t} > 0 \quad \text{if} \quad N_1^* < N < N_2^* \tag{2.70}$$

In turn, the fish population tends to extinction if $N(0) < N_1^*$, and tends to N_2^* if $N(0) > N_1^*$. Therefore, the equilibrium point P_2 is globally asymptotically stable in the region

$$\left\{(N, C, E): \quad N_1^* \le N \le K(0), \quad 0 \le C \le \frac{Q}{\delta}, \quad 0 \le E \le \frac{\eta}{\eta_0}\left(\frac{Q}{\delta} - C_{\text{lim}}\right)\right\} \tag{2.71}$$

For the parameter value of Table 2.11, the equilibrium points are [14]

$$N_1^* = 6{,}671; \quad N_2^* = 75{,}069 \tag{2.72}$$

The code `fish_odes` implements a solution to Eqs. (2.58)–(2.62) including several variations. First, the function to define the model equations is listed.

```
function xt = fish_odes(t,x)
```

```
% Set global variables
global r0 Clim alpha Cdeath K0 Klim beta H Q delta delta0
global eta eta0

% x has three columns corresponding to three different
% column solution vectors (which can be used for
% numerical evalution of the Jacobian):
ncols = size(x,2);

for j=1:ncols

    % Transfer dependent variables
    N = x(1,j);
    C = x(2,j);
    E = x(3,j);

    % Temporal derivatives
    if C < Clim
        r = r0;
    elseif Clim <= C < Cdeath
        r = r0-alpha*(C-Clim);
    else
        r = 0;
    end
    %
    if C < Clim
        K = K0;
    elseif Clim <= C < Cdeath
        K = K0-beta*(C-Clim);
    else
        K = Klim;
    end

    Nt  = r*N - r0*N^2/K - H;
    Ct  = Q - delta*C - delta0*E;
    Et = eta*(C-Clim) - eta0*E;

    % Transfer temporal derivatives
    % (One column for each column of x)
    xt(:,j) = [Nt Ct Et]';
end;
```

Function fish_odes Function for Eqs. (2.58)–(2.62)

We can note the following points about `fish_odes`.

1. After defining a set of global variables, a $3\times$ `ncol` matrix, `x(3,1)`, is defined, where the first index defines a row dimension of three for the state vector $(N, C, E)^T$, and the second index defines a column dimension of one for one computed solution corresponding to a single set of parameters and initial conditions defined in the main program that calls `fish_odes` (discussed next)

```
% x has three columns corresponding to three different
% column solution vectors
ncols = size(x,2);
% Step through the ncol solutions
for j=1:ncols
    %
    % Transfer dependent variables
    N = x(1,j);
```

```
C = x(2,j);
E = x(3,j);
```

2. The expressions (2.59) to (2.60) are then programmed

```
% Parameters r, K set
    %
    if C < Clim
        r = r0;
    elseif Clim <= C < Cdeath
        r = r0-alpha*(C-Clim);
    else
        r = 0;
    end
    %
    if C < Clim
        K = K0;
    elseif Clim <= C < Cdeath
        K = K0-beta*(C-Clim);
    else
        K = Klim;
    end
```

Note in particular the switching to three possible values of the parameters r and k depending on the current value of C.

3. The temporal derivatives for the three solutions are then computed according to ODEs, (2.58), (2.61), and (2.62)

```
% Temporal derivatives
Nt  = r*N - r0*N^2/K - H;
Ct  = Q - delta*C - delta0*E;
Et = eta*(C-Clim) - eta0*E;
```

4. The temporal derivatives are then stored in a 3×1 matrix for return to the ODE integrator (a total of $3 \times 1 = 3$ derivatives)

```
% Transfer temporal derivatives
% (One column for each column of x)
xt(:,j) = [Nt Ct Et]';
```

Note the index for the solutions, `j`, is incremented by the `for` statement at the beginning of `fish_odes` that is terminated by the `end` statement. In the present case, `ncol=1` (just one solution is computed). However, this approach to computing multiple solutions in parallel could be used to conveniently compare solutions, generated, for example, for multiple sets of initial conditions or model parameters. In other words, having the solutions available simultaneously would facilitate their comparison, e.g., by plotting them together. This feature of computing and plotting multiple solutions in a parametric study is illustrated subsequently in Fig. 2.11.

The main program that calls `fish_odes` is presented in `fish_main`.

```
close all
clear all

% start a stopwatch timer
tic

% set global variables
global r0 Clim alpha Cdeath K0 Klim beta H Q delta delta0
global eta eta0 fac thresh vectorized

% model parameters
r0     = 0.02;
Clim   = 50;
alpha  = 0.0001;
Cdeath = (r0+alpha*Clim)/alpha;
K0     = 100000;
Klim   = 100;
beta   = 0.05;
H      = 100;
Q      = 2;
delta  = 0.002;
delta0 = 0.005;
eta0   = 0.004;

% select the control action 'eta' (comment/decomment one
% of the actions)
 action = 'strong'
% action = 'moderate'
% action = 'weak'
switch action
    % strong
    case('strong')
        eta = 0.5;
    % moderate
    case('moderate')
        eta = 0.1;
    % weak
    case('weak')
        eta = 0.04;
end

% equilibrium points
[N1star,N2star] = equilibrium_fish(r0,Clim,alpha,...
                    Cdeath,K0,Klim,beta,H,Q,delta,...
                    delta0,eta,eta0)

% initial conditions
t0 = 0;
tf = 3560;
N  = 72500;
%     N = 5000;
C = 80;
E = 190;
x = [N C E]';

% call to ODE solver (comment/decomment one of the methods
% to select a solver)
```

```
%      method = 'euler'
%      method = 'midpoint'
%      method = 'heun'
%      method = 'heun12'
%      method = 'rk4'
%      method = 'rkf45'
%      method = 'ros3p'
method = 'ros23p'
%      method = 'ode45'
%      method = 'ode15s'
%      method = 'lsodes'
switch method
    % Euler
    case('euler')
        Dt     = 0.1;
        Dtplot = 20;
        [tout,xout] = euler_solver(@fish_odes,t0,tf,x,...
                                   Dt,Dtplot);
    % midpoint
    case('midpoint')
        Dt     = 5;
        Dtplot = 20;
        [tout,xout] = midpoint_solver(@fish_odes,t0,...
                                      tf,x,Dt,Dtplot);
    % Heun
    case('heun')
        Dt     = 5;
        Dtplot = 20;
        [tout,xout] = heun_solver(@fish_odes,t0,tf,x,...
                                  Dt, Dtplot);
    % Heun12
    case('heun12')
       hmin      = 0.0001;
       nstepsmax = 1e5;
       abstol    = 1e-3;
       reltol    = 1e-3;
       Dtplot    = 20;
       [tout,xout] = heun12_solver(@fish_odes,t0,tf,...
                                   x,hmin,nstepsmax,...
                                   abstol,reltol,Dtplot);
    % rk4
    case('rk4')
        Dt     = 5;
        Dtplot = 20;
        [tout, xout] = rk4_solver(@fish_odes,t0,tf,x,...
                                  Dt,Dtplot);
    % rkf45
    case('rkf45')
        hmin      = 0.0001;
        nstepsmax = 1e5;
        abstol    = 1e-3;
        reltol    = 1e-3;
        Dtplot    = 20;
        [tout,xout,eout] = rkf45_solver(@fish_odes,t0,...
                                  tf,x,hmin,nstepsmax,...
                                  abstol,reltol,Dtplot);
        figure(2)
        plot(tout/356,eout(:,1),':r');
        hold on
```

```
    plot(tout/356,eout(:,2),'--b');
    plot(tout/356,eout(:,3),'--g');
    xlabel('t');
    ylabel('e(t)');
% ros3p
case('ros3p')
    Dt     = 1;
    Dtplot = 20;
    fac    = [];
    thresh = 1e-12;
    [tout,xout] = ros3p_solver(@fish_odes,...
                               @jacobian_num_fish,...
                               t0,tf,x,Dt,Dtplot);
% ros23p
case('ros23p')
    hmin      = 0.0001;
    nstepsmax = 1e5;
    abstol    = 1e-3;
    reltol    = 1e-3;
    Dtplot    = 1;
    fac       = [];
    thresh    = 1e-12;
    [tout,xout,eout] = ros23p_solver(@fish_odes,...
                               @jacobian_num_fish,...
                               @ft_fish,t0,tf,x,...
                               hmin,nstepsmax,...
                               abstol,reltol,Dtplot);
    figure(2)
    plot(tout,eout(:,1),':r');
    hold on
    plot(tout,eout(:,2),'--b');
    plot(tout,eout(:,3),'--g');
    xlabel('t');
    ylabel('e(t)');
% ode45
case('ode45')
    options = odeset('RelTol',1e-3,'AbsTol',1e-3);
    Dtplot = 20;
    t = [t0:Dtplot:tf];
    [tout,xout] = ode45(@fish_odes,t,x,options);
% ode15s
case('ode15s')
    options = odeset('RelTol',1e-3,'AbsTol',1e-3);
    Dtplot=20;
    t=[t0:Dtplot:tf];
    [tout, xout] = ode15s(@fish_odes,t,x,options);
% lsodes
case('lsodes')
    resname = 'fish_odes';
    jacname = '[]';
    neq     = 3;
    Dtplot  = 20;
    tlist   = [t0+Dtplot:Dtplot:tf];
    itol    = 1;
    abstol  = 1e-3;
    reltol  = 1e-3;
    itask   = 1;
    istate  = 1;
    iopt    = 0;
```

```
        lrw     = 50000;
        rwork   = zeros(lrw,1);
        liw     = 50000;
        iwork   = zeros(liw,1);
        mf      = 222;
        [tout,xout] = lsodes(resname,jacname,neq,x,t0,...
                             tlist,itol,reltol,abstol,...
                             itask,istate,iopt,rwork,...
                             lrw,iwork,liw,mf);
end

% plot results
figure(1)
subplot(3,1,1)
plot(tout/356,xout(:,1),'-');
%     xlabel('t [years]');
ylabel('N(t)');
%     title('Fish population');
subplot(3,1,2)
plot(tout/356,xout(:,2),'-');
%     xlabel('t [years]');
ylabel('C(t)');
%     title('Acid concentration')
subplot(3,1,3)
plot(tout/356,xout(:,3),'-');
xlabel('t [years]');
ylabel('E(t)');
%     title('Liming effort');

% read the stopwatch timer
tcpu=toc;
```

Script fish_main Main program that calls subordinate routine `fish_odes`

We can note the following points about `fish_main`.

1. First, global variables are defined and the model parameters are set.
2. An integrator is then selected from a series of basic or advanced integrators (also including LSODES, which is a ODE solver from ODEPACK [2] transformed into a MEX-file - we will give more details on how to create MEX-files in a subsequent section) by uncommenting a line, in this case for `ros23p_solver` discussed previously.
3. The degree of liming is then set by a switch function which in this case selects a character string for "strong."
4. Calls to the various integrators are listed. The call to `ros23p` is similar to that of the main program `Budworm_main`. The details for calling the various integrators are illustrated with the coding of the calls. Table 2.12 shows a comparison of the performance of these solvers. Note that the performance of the solvers increases as the control law becomes weaker, in particular, for Euler's and Heun's methods.
5. The numerical solutions resulting from the calls to the various integrators are plotted. Note in the case of `ros23p`, the estimated errors (given by the sub-

Table 2.12 Performance of different IVP solvers applied to the liming to remediate acid rain problem

Control action		Euler	Heun	rkf45	ros23p	ode45	ode15s Adams	ode15s BDF
Strong	N. steps	17,800	1,780	212	213	101	281	271
	CPU time	38.9	8.4	3.4	5.0	2.4	4.4	4.2
Moderate	N. steps	3560	356	193	203	44	123	127
	CPU time	8.0	1.8	3.2	4.9	1.4	2.8	2.6
Weak	N. steps	1780	178	193	193	29	79	82
	CPU time	4.0	1.0	3.2	4.6	1.2	2.2	2.1

Computational times are normalized with respect to the time spent by the most efficient solver, in this example, Heun's solver with weak control law

traction of Eq. (2.45a) and (2.45b) and implemented in `ssros23p`) are also plotted.

To complete the programming of Eqs. (2.58)–(2.62), we require a function for the calculation of the equilibrium points (called by `fish_odes`) and a function for the Jacobian matrix (called by `ssros23p`). Function `equillibrium_fish` is a straightforward implementation of Eqs. (2.64)–(2.67).

```
function [N1star,N2star] = equilibrium_fish(r0,Clim,...
                           alpha,Cdeath,K0,Klim,beta,...
                           H,Q,delta,delta0,eta,eta0)
%
Estar = eta*(Q/delta-Clim)/(eta0+eta*(delta0/delta));
Cstar = (Q-delta0*Estar)/delta;

%
if Cstar < Clim
    r = r0;
elseif Clim <= Cstar < Cdeath
    r = r0-alpha*(Cstar-Clim);
else
    r = 0;
end

%
if Cstar < Clim
    K = K0;
elseif Clim <= Cstar < Cdeath
    K = K0-beta*(Cstar-Clim);
else
    K = Klim;
end

%
N1star = (K*r)/(2*r0)*(1-sqrt(1-(4*r0*H)/(K*r^2)));
N2star = (K*r)/(2*r0)*(1+sqrt(1-(4*r0*H)/(K*r^2)));
```

Function equillibrium_fish Function for the calculation of the equilibrium points from Eq. (2.64) to (2.67)

Function `jacobian_fish` computes the analytical Jacobian of the ODE system, Eqs. (2.58), (2.61) and (2.62)

```
function Jac = jacobian_fish(t,x)
%
% Set global variables
global r0 Clim alpha Cdeath K0 Klim beta H Q delta delta0
global eta eta0
%
% Transfer dependent variables
N = x(1);
C = x(2);
E = x(3);
%
% Jacobian matrix
%
%     Nt  = r*N - r0*N^2/K - H;
%     Ct  = Q - delta*C - delta0*E;
%     Et = eta*(C-Clim) - eta0*E;
%
%
if C < Clim
    r = r0;
    K = K0;
    Jac = [r-2*r0*N/K    0         0       ;
           0            -delta    -delta0 ;
           0            eta       -eta0   ];
elseif Clim <= C < Cdeath
    r = r0-alpha*(C-Clim);
    K = K0-beta*(C-Clim);
    Jac = [r-2*r0*N/K   -alpha*N+beta*r0*N^2/K^2   0       ;
           0            -delta                     -delta0 ;
           0            eta                        -eta0   ];
else
    r = 0;
    K = Klim;
    Jac = [r-2*r0*N/K    0         0       ;
           0            -delta    -delta0 ;
           0            eta       -eta0   ];
end
```

Function jacobian_fish Function for the calculation of the Jacobian matrix of Eqs. (2.58), (2.61) and (2.62)

Note that the Jacobian matrix programmed in `jacobian_fish` also has a set of three switching functions, which give different elements of the Jacobian matrix depending on the current value of C. To illustrate the origin of the elements of the Jacobian matrix, consider the first case for $C < C_{\text{lim}}$.

```
if C < Clim
    r = r0;
    K = K0;
Jac = [r-2*r0*N/K    0         0  ;
         0        -delta    -delta0 ;
         0          eta      -eta0  ];
```

The first row has the three elements for the Jacobian matrix from Eq. (2.58). The RHS function of Eq. (2.58) is:

$$r(C)N - \frac{r_0 N^2}{K(C)} - H$$

with $r = r_0$, $K = K_0$, the RHS function becomes

$$r_0 N - \frac{r_0 N^2}{K_0} - H$$

We now have the following expressions for the derivative of this function with respect to each of the state variables

$$\frac{\partial\left(r_0 N - \frac{r_0 N^2}{K_0} - H\right)}{\partial N} = r_0 - \frac{2 r_0 N}{K_0}; \qquad \frac{\partial\left(r_0 N - \frac{r_0 N^2}{K_0} - H\right)}{\partial C} = 0$$

$$\frac{\partial\left(r_0 N - \frac{r_0 N^2}{K_0} - H\right)}{\partial E} = 0$$

which are programmed as

```
Jac = [r-2*r0*N/K     0          0        ;
```

The remaining six elements of the Jacobian matrix follow in the same way from the RHS functions of Eqs. (2.61) and (2.62).

We can note three important features of the programming of the Jacobian matrix:

- If the state vector is of length n (i.e., n ODEs), the Jacobian matrix is of size $n \times n$. This size grows very quickly with increasing n. Thus for large systems of ODEs, e.g., $n > 100$, the Jacobian matrix is difficult to evaluate analytically (we must derive $n \times n$ derivatives).
- The Jacobian matrix can have a large number of zeros. Even for the small 3×3 ODE problem of Eqs. (2.58), (2.61), and (2.62), the Jacobian matrix has four zeros in a total of nine elements. Generally, for physical problems, the fraction of zeros increases rapidly with n and is typically 0.9 or greater. In other words, Jacobian matrices tend to be *sparse*, and algorithms that take advantage of the sparsity by not storing and processing the zeros can be very efficient. Therefore, scientific computation often depends on the use of sparse matrix algorithms, and scientific software systems such as MATLAB have sparse matrix utilities.
- Because the differentiation required for an analytical Jacobian is often difficult to develop, numerical methods for producing the required $n \times n$ partial derivatives are frequently used, as illustrated with Eq. (2.49) and in function `jacobian_stiff_odes_fd`. A routine for calculating the numerical Jacobian for Eqs. (2.58), (2.61), and (2.62) is presented in function `jacobian_num_fish`.

```
function Jac = jacobian_num_fish(t,x)
%
% Set global variables
global fac thresh vectorized
%
% numerical Jacobian matrix
tstar = t;
xstar = x;
xtstar = fish_odes(tstar,xstar);
threshv = [thresh;thresh;thresh];
vectorized = 1;
[Jac, fac] = numjac(@fish_odes,tstar,xstar,xtstar,threshv,...
                    fac,vectorized);
```

Function jacobian_num_fish Function for the numerical calculation of the Jacobian of Eqs. (2.58), (2.61) and (2.62)

We will not consider the details of `jacobian_num_fish`, but rather, just point out the use of `fish_odes` for ODEs (2.58), (2.61) and (2.62), and the MATLAB routines `threshv` and `numjac` for the calculation of a numerical Jacobian. Clearly calls to the MATLAB routines for a numerical Jacobian will be more compact than the programming of the finite difference approximations for large ODE systems (as illustrated by the small 2×2 ODE system in function jacobian_stiff_odes_fd).

Coding to call `jacobian_num_fish` is illustrated in the use of `ros3p` (fixed step LIRK integrator) in `fish_main`.

This completes the coding of Eqs. (2.58), (2.61) and (2.62). We conclude this example by considering the plotted output from the MATLAB code `fish_main` given in Figs. 2.9 and 2.10. We note in Fig. 2.9 that the solution $N(t)$ reaches a stable equilibrium point $N_2^* = 93,115$, which is the value for strong liming ($\eta = 0.5$) as set in `fish_main`. This constrasts with the stable equilibrium point of Eq. (2.72) for weak liming ($\eta = 0.04$). As might be expected, the equilibrium point for weak liming is below that for strong liming (increased liming results in a higher equilibrium fish population).

Figure 2.10 indicates that the maximum estimated error for $N(t)$ is about 3.6. Since $N(t)$ varies between 72,500 (the initial condition set in fish_main) and 93,115 (the final equilibrium value of $N(t)$), the maximum fractional error is 3.6/72500 = 0.0000496, which is below the specified relative error of `reltol=1e-3` set in `fish_main`. Of course, this error of 3.6 is just estimated, and may not be the actual integration error, but this result suggests that the numerical solution has the specified accuracy.

Also, the estimated errors for the other two state variables, $C(t)$ and $E(t)$, are so small they are indiscernible in Fig. 2.10. This is to be expected since the magnitudes of these variables are considerably smaller than for $N(t)$ (see Fig. 2.9). In conclusion, these results imply that the error estimate of the embedded LIRK algorithm performed as expected in this application.

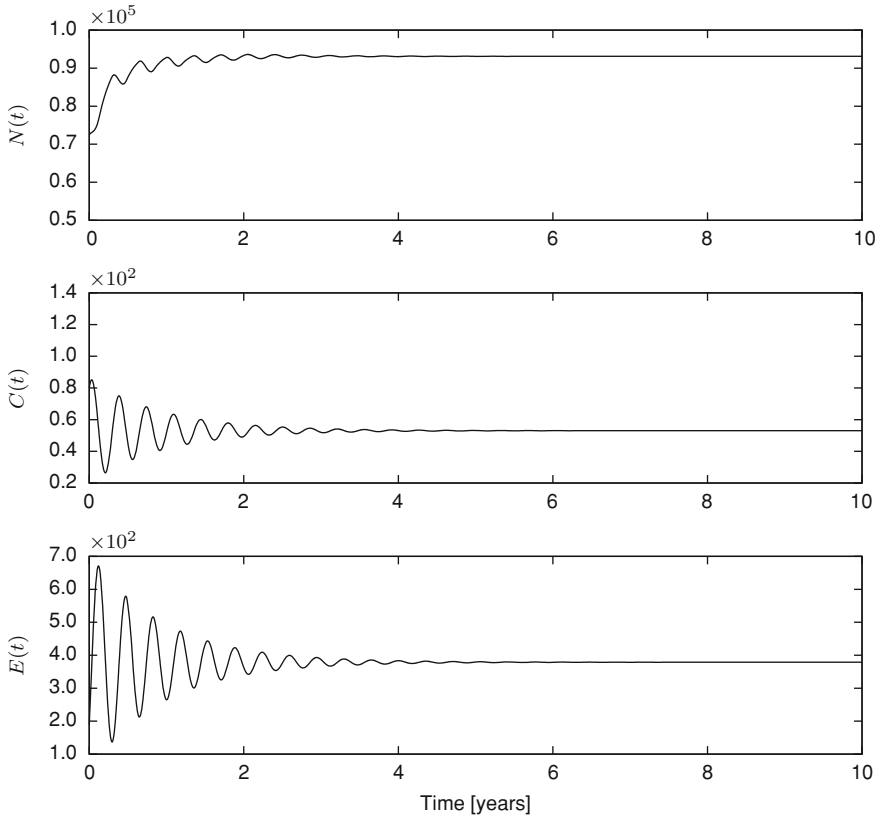

Fig. 2.9 Solution of Eqs. (2.58), (2.61) and (2.62) from `fish_main` for a strong control action $\eta = 0.5$

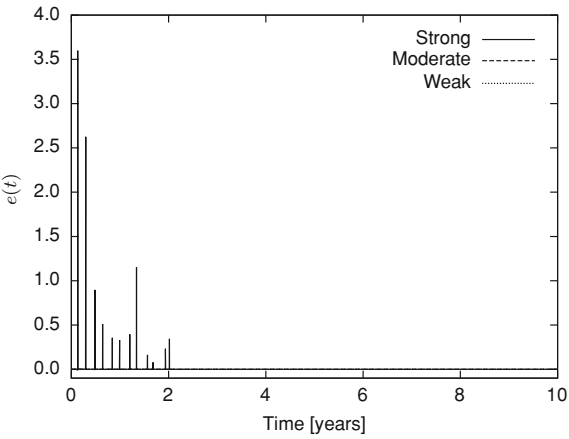

Fig. 2.10 Estimated error from `ros23p` for Eqs. (2.58), (2.61) and (2.62) from fish_main

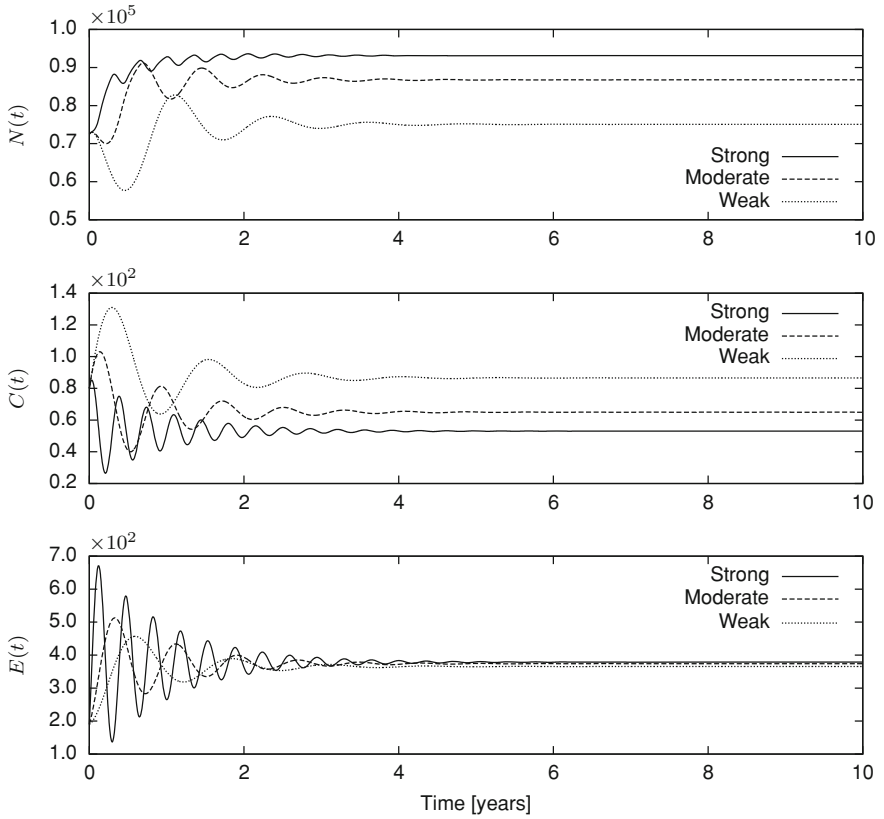

Fig. 2.11 State variable plots for Eqs. (2.58), (2.61) and (2.62) for the three liming conditions corresponding to $\eta = 0.04, 0.1, 0.5$

We conclude this discussion by including a few plots—see Fig. 2.11—with the solutions for the three liming conditions ($\eta = 0.04, 0.1, 0.5$) superimposed so that the effect of the liming is readily apparent.

As another result, we can mention that for the weak liming condition ($\eta = 0.04$), if the initial fish population is lower than the first unstable equilibrium point $N_1^* = 6{,}661$, e.g. $N_1^* = 5{,}000$, the fish population decreases and eventually vanishes as predicted by the theoretical analysis. Also, for this case, an event detection is necessary to prevent the fish population from becoming negative (a function `events.m` monitors the value of $N(t)$). In this respect, the model equations are not well formulated (they allow $N(t)$ to become negative).

2.6 On the Use of SCILAB and OCTAVE

As mentioned before, MATLAB has been selected as the main programming environment because of its convenient features for vector/matrix operations that are central to the solution of AE/ODE/PDE systems. In addition, MATLAB provides a very complete library of numerical algorithms (for numerical integration, matrix operations, eigenvalue computation,…), e.g., the MATLAB ODE SUITE [1], that can be used advantageously in combination with the proposed MOL toolbox.

However, there exist very powerful open source alternatives, such as SCILAB (for Scientific Laboratory) or OCTAVE, that can be used for the same purposes. SCILAB and OCTAVE provide high-level, interpreted programming environments, with matrices as the main data type, similar to MATLAB.

Initially named Ψlab (Psilab), the first software environment was created in 1990 by researchers from INRIA and *École nationale des ponts et chaussées (ENPC)* in France. The SCILAB Consortium was then formed in 2003 [15] to broaden contributions and promote SCILAB in academia and industry. In 2010, the Consortium announced the creation of SCILAB Enterprises, which develops and maintain SCILAB as an open source software but proposes commercial services to industry (professional software and project development). With regard to the solution of ordinary differential equations, SCILAB has an interface, called *ode*, to several solvers, especially those from the FORTRAN library ODEPACK originally developed by Alan Hindmarsh [2].

OCTAVE [16] was initially developed by John Eaton in the early 1990s, and then developed further by many contributors following the terms of the GNU General Public License (GPL) as published by the Free Software Foundation. The name OCTAVE is inspired by Octave Levenspiel, former Chemical Engineering Professor of John Eaton, who was known for his ability to solve numerical problems. OCTAVE has a built-in ODE solver based on LSODE [3], and a DAE solver, DASSL, originally developed by Linda Petzold [4]. Additional contributed packages are also available, e.g., OdePkg, which has a collection of explicit and implicit ODE solvers and DAE solvers.

An interesting feature of SCILAB and OCTAVE, as we will see, is the degree of compatibility with MATLAB codes. In many cases, slight modifications of the MATLAB codes will allow us to use them in SCILAB or OCTAVE.

Let us now show the programming of the spruce budworm dynamics—see Eqs. (2.50)–(2.55)—in SCILAB highlighting the main differences with MATLAB.

The SCILAB script `spruce_budworm_main.sci` is the main program, which now has an extension .sci instead of .m. The main features of this program are the following:

- Commented lines (these lines are not read by the interpreter but are very useful to explain the code) in SCILAB start with a double slash (//) as in C++, which is the equivalent to the percentage symbol (%) in MATLAB.

- The information to be displayed during the execution of the program can be changed in SCILAB with the `mode(k)` command. With `k=-1` the code runs silently (no information is displayed).
- The stopwatch timer commands coincide with the MATLAB ones (`tic`, `toc`)
- Setting the global variables in SCILAB is slightly different than in MATLAB. In SCILAB, each variable is inside quotation marks and it is separated from the other variables by commas.
- One important difference with respect to the MATLAB codes is that the functions must be loaded with the `exec` command. For instance, the ODEs are programmed in function `spruce_budworm_odes.sci` and it is loaded with the command `exec('spruce_budworm_odes.sci')`. In MATLAB any function defined in the path can be called.
- The definition of model parameters and initial conditions is exactly the same in MATLAB and SCILAB.
- After the definition of the initial conditions, the ODE solver is chosen. In this case two possibilities are offered, namely, *Euler's* method and a *Runge–Kutta–Fehlberg* implementation. Note that all our basic time integrators (presented in this chapter) can easily be translated to SCILAB, and are provided in the companion software. In the application example, *Euler's* method is chosen by commenting out the line `method = 'rkf45'`. Then the `select` command with two different `cases` is used. This command is equivalent to the `switch` command in MATLAB. As `'Euler'` is the method of choice, the code will run the lines corresponding to this selection while the lines corresponding to `'rkf45'` will be blind to the execution. An example of use of the SCILAB ODE library is included in Sect. 3.12.
- In order to use the Euler solver, it is required to load the function that implements it: `exec('euler_solver.sci')`. Then the function where the ODEs are defined is called (`spruce_budworm_odes.sci`) practically in the same way as in MATLAB. In this sense, in SCILAB we have

```
[tout,yout] = euler_solver(''spruce_budworm_odes(t,x)'',...
                           t0,tf,x,Dt,Dtplot);
```

  while in MATLAB this line is written as:

```
[tout,yout] = euler_solver(@spruce_budworm_odes,...
                           t0, tf, x, Dt, Dtplot);
```

- Finally, the solution is plotted using the same commands as in MATLAB

```
// Display mode
mode(-1);

// Clear previous workspace variables
clear

// start a stopwatch timer
```

```
tic

// set global variables
global("rb","k","pbeta","a","rs","Ks","re","Ke","p","Te")

// Load the subroutines
exec('spruce_budworm_odes.sci')

// model parameters
rb    = 1.52;
k     = 355;
pbeta = 43200;
a     = 1.11;
rs    = 0.095;
Ks    = 25440;
re    = 0.92;
Ke    = 1;
p     = 0.00195;
Te    = 0.03;

// initial conditions
t0 = 0;
tf = 200;
B  = 10;
S  = 7000;
E  = 1;
x  = [B,S,E]';

// call to ODE solver (comment/decomment one of the methods
// to select a solver)
method = 'Euler'
//method = 'rkf45'

select method,
    case 'Euler' then
        // Load the Euler solver subroutine
        exec('euler_solver.sci')
        Dt          = 0.01;
        Dtplot      = 0.5;
        [tout,yout] = euler_solver("spruce_budworm_odes(t,x)",...
                                   t0,tf,x,Dt,Dtplot);
    case 'rkf45' then
        // Load the rkf45 solver subroutines
        exec('rkf45_solver.sci')
        exec('ssrkf45.sci')
        hmin      = 0.001;
        nstepsmax = 1000;
        abstol    = 0.001;
        reltol    = 0.001;
        Dtplot    = 0.5;
        [tout,yout,eout] =...
            rkf45_solver("spruce_budworm_odes(t,x)",t0,tf,x,...
                         hmin,nstepsmax,abstol,reltol,Dtplot);
end

// Plot the solution
subplot(3,1,1)
plot(tout,yout(:,1))
ylabel('B(t)','FontSize',2);
```

```
subplot(3,1,2)
plot(tout,yout(:,2))
ylabel('S(t)','FontSize',2);
subplot(3,1,3)
plot(tout,yout(:,3))
xlabel('Time [years]','FontSize',2);
ylabel('E(t)','FontSize',2);

// read the stopwatch timer
tcpu = toc();
```

Script spruce_budworm_main.sci Main program that calls functions `Euler_solver.sci` and `spruce_budworm_odes.sci`

The other two SCILAB functions required to solve the problem are: `spruce_budworm_odes.sci` (where the RHS of the ODEs are defined) and `Euler_solver.sci` (containing the implementation of the Euler solver). These codes are practically the same as in MATLAB, the main differences are those already mentioned in the main script, i.e., the symbol used to comment the lines (//), the way of setting the global variables and the `mode` command to select the information printed during execution.

```
// Display mode
mode(-1);

function [xt] = spruce_budworm_odes(t,x)

// Output variables initialisation (not found in input
// variables)
xt=[];

// Set global variables
global("rb","k","pbeta","a","rs","Ks","re","Ke","p","Te")

// Transfer dependent variables
B = x(1);
S = x(2);
E = x(3);

// Model Parameters
Kb    = ((k*S)*(E^2))/(E^2+Te^2);
alpha = a*S;
g     = (pbeta*(B^2))/(alpha^2+B^2);
P     = (p*(E^2))/(Te^2+E^2);

// Temporal derivatives
Bt = (rb*B)*(1-B/Kb) - g;
St = (rs*S)*(1-(S/Ks)*(Ke/E));
Et = ((re*E)*(1-E/Ke)-(P*B)/S);

// Transfer temporal derivatives
xt = [Bt,St,Et]';
endfunction
```

Function spruce_budworm_odes.sci Right hand side of the ODE system (2.50)–(2.55).

Finally, the SCILAB function `Euler_solver.sci` contains another difference with respect to MATLAB: the `feval` command

```
xnew = x + feval(odefunction,t,x)*Dt;
```

is substituted by

```
xnew = x+evstr(odefunction)*Dt;
```

where `odefunction` is an input parameter of type *string*.

```
// Display mode
mode(-1);

function [tout,xout] = euler_solver(odefunction,t0,tf,...
                                    x0,Dt,Dtplot)

// Output variables initialisation (not found in input
// variables)
tout=[];
xout=[];

// Initialization
plotgap = round(Dtplot/Dt);// number of computation
// steps within a plot interval
Dt      = Dtplot/plotgap;
nplots = round((tf-t0)/Dtplot);// number of plots
t       = t0;// initialize t
x       = x0;// initialize x
tout    = t0;// initialize output value
xout    = x0';// initialize output value

// Implement Euler''s method
for i = 1:nplots
  for j = 1:plotgap
    // Use MATLAB''s feval function to access the
    // function file, then take Euler step
    xnew = x+evstr(odefunction)*Dt;
    t = t+Dt;
    x = xnew;
  end;
  //   Add latest result to the output arrays
  tout = [tout;t];
  xout = [xout;x'];
  //
end;
endfunction
```

Function Euler_solver.sci SCILAB version of the basic Euler ODE integrator

As mentioned before, SCILAB has an interface, called *ode*, to several solvers. The use of function `ode` in SCILAB will be illustrated with a bioreactor example (see Sect. 3.12). The different options for the solvers include: `lsoda` (that automatically

selects between Adams and BDF methods), adaptive RK of order 4, and Runge–Kutta–Fehlberg.

On the other hand, OCTAVE comes with LSODE (for solving ODEs) and DASSL, DASPK and DASRT (designed for DAEs). However, a whole collection of around 15 solvers is included in the package OdePkg http://octave.sourceforge.net/odepkg/overview.html. This package include explicit RK solvers of different orders for ODE problems, backward Euler method and versions of different FORTRAN solvers for DAEs (as RADAU5, SEULEX, RODAS, etc.), as well as more sophisticated solvers for implicit differential equations and delay differential equations which are out of the scope of this book.

It must be mentioned that OCTAVE is even more compatible with MATLAB than SCILAB. The main difference between MATLAB and OCTAVE is the call to the ODE solvers when built-in functions are used. If time integration is carried out using our solvers (Euler, rkf45, ros23p, etc.), the MATLAB codes developed in this chapter can be directly used in OCTAVE. However, when built-in solvers are used, slight modifications are required. For instance, the call to ode15s in MATLAB for the spruce budworm problem is of the form:

```
options = odeset('RelTol',1e-6,'AbsTol',1e-6);
t       = [t0:0.5:tf];
[tout, yout] = ode15s(@spruce_budworm_odes,t,x,options);
```

where `spruce_budworm_odes` is the name of the function where the RHS of the ODE equations is implemented, `t` is the time span for the integration, `x` are the initial conditions and `options` is an optional parameter for setting integration parameters as the tolerances or the maximum step size.

The call to the lsode solver in OCTAVE is carried out as follows:

```
lsode_options('absolute tolerance',1e-6);
lsode_options('relative tolerance',1e-6);
tout               = [t0:0.5:tf];
[yout, istat, msg] = lsode(@spruce_budworm_odes, x, tout);
```

The most important difference with respect to MATLAB is the order of the dependent and independent problem variables x and t. Note that in OCTAVE x is the second input parameter and t the third. This also affects function `spruce_budworm_odes`. In MATLAB the first line of this code reads as:

```
function xt = spruce_budworm_odes(t,x)
```

while in OCTAVE we have

```
function xt = spruce_budworm_odes(x,t)
```

Note that if we want to reuse this code for integration with our solvers, they must be slightly modified accordingly. For instance, in MATLAB, Euler solver make calls to function `spruce_budworm_odes`

```
xnew = x + feval(odefunction,t,x)*Dt;
```

Table 2.13 Performance of different IVP solvers in several environments for the spruce budworm and liming to remediate acid rain problems

		Spruce budworm	Acid rain
MATLAB	Euler	23.85	2.96
	rkf45	10.00	2.53
	ode45	10.77	1.77
	ode15s	21.92	3.27
SCILAB	Euler	117.31	224.61
	rkf45	24.23	9.61
	lsode (Adams)	10.38	5.00
	lsode (BDF)	13.46	7.69
OCTAVE	Euler	43.85	52.30
	rkf45	19.23	5.00
	lsode	6.15	1.00
	dassl	13.46	1.77

In the acid rain problem a *strong* control law, which is the most challenging, is used. Times have been normalized with respect to the most efficient case, i.e., LSODE in OCTAVE for the acid rain problem

In OCTAVE this must be modified to

```
xnew = x + feval(odefunction,x,t)*Dt;
```

To conclude this section, a comparison of the performance of different solvers in MATLAB, SCILAB, and OCTAVE is included in Table 2.13. When our simple IVP solvers are used, i.e., euler and rkf45, MATLAB is by far the most efficient environment for both problems. The computational cost in OCTAVE is clearly the largest one. On the other hand, when built-in solvers are used, OCTAVE is the most efficient alternative (especially with LSODE) while SCILAB and MATLAB computational costs are of comparable magnitude (SCILAB is more efficient than MATLAB for solving the spruce budworm problem and the reverse is true when solving the acid rain problem). However, it should be noted that in general, when the size and complexity of the problem increases, MATLAB will usually appear as the most efficient alternative even when using built-in solvers.

2.7 How to Use Your Favorite Solvers in MATLAB?

This last section is intended for the reader with some background knowledge in programming, and we suggest for the reader who is not interested in implementation details to skip this material, and possibly to come back to it later on.

MATLAB EXecutable files (MEX-files) are dynamically linked subroutines produced from C or FORTRAN source code that, when compiled, can be run within MATLAB in the same way as MATLAB M-files or built-in functions www.mathworks.com/support/tech-notes/1600/1605.html.

The main reasons for using MEX-files are listed below:

1. During the last few decades, a huge collection of C and FORTRAN numerical algorithms have been created and largely tested by different research and industrial organizations. Such codes cover a large variety of fields such as linear algebra, optimization, times series analysis among many others. Particularly interesting for this chapter is the collection of efficient and reliable ODE solvers. Using the MEX-files we can call these subroutines without the need of rewriting them in MATLAB.
2. MATLAB is a high-level language and as such the programming is easier than in low-level languages as C/C++ or FORTRAN. As a drawback, MATLAB codes are, in general, slower than C and FORTRAN. MEX-files allow us to increase the efficiency of MATLAB M-files or built-in functions.

In order to create a MEX-file, first a gateway routine is required. The gateway is the routine through which MATLAB accesses the rest of the routines in MEX-files. In other words, it is the connection bridge between MATLAB and the FORTRAN or C subroutines. The standard procedure for creating gateways is described in http://www.mathworks.com/support/tech-notes/1600/1605.html?BB=1. This procedure is, however, tiresome, prone to mistakes and with a complex debugging only recommendable for those with good programming skills.

There is however a tool (OPENFGG) for generating semiautomatically the gateways. This tool can be downloaded from the url http://www8.cs.umu.se/~dv02jht/exjobb/download.html and includes a graphical user interphase. There are four steps to create the gateway with this tool:

- Create a new project *File* → *New*
- Add the source files to parse. In this step we include the FORTRAN code for which we want to generate the mex file (e.g. `mycode.f`)
- Select the inputs and the outputs of `mycode.f`
- Press the *Compile* buttom. The gateway `mycodegw.f` is generated

One of the main drawbacks of OPENFGG is that it does not allow to create reverse gateways, this is, codes for calling MATLAB files from FORTRAN.

It must be noted that for creating a MEX file an adequate FORTRAN or C/C++ compiler must be installed.

Now we are ready to create the MEX-file. To that purpose we open a MATLAB session and type:

```
mex -setup
```

And choose one of the installed compilers. This step is only required to be done once. After this, type

```
mex mycode.f mycodegw.f
```

Which creates the MEX-file that can be used at the MATLAB command prompt in the same way as any M-file or built-in function.

From the MATLAB version 7.2 (R2006a), it is possible to use open source FORTRAN and C compilers (g95, gfortran, gcc) through the tool GNUMEX. GNUMEX

allows to set up the Windows versions of gcc (also gfortran and g95) to compile mex files. The procedure is described in detail at http://gnumex.sourceforge.net, However, the main steps are summarized below:

- Install the packages for the gcc compiler. The easiest way is to Download Mingw from http://www.mingw.org/ and install it. The installation path `C:\mingw` is recommended.
- Download the g95 compiler from http://www.g95.org/downloads.shtml and install it. The option *Self-extracting Windows x86 (gcc 4.1, experimental)* is recommended.
- Download GNUMEX http://sourceforge.net/projects/gnumex/ unzip the files and place the folder in a directory (e.g `C:\gnumex`).
- Open a MATLAB session, go to the path where gnumex was unzipped and type `gnumex`. A window will open. Through this window we will modify the `mexopts.bat` which contains the information about the compilers and options used for creating the MEX-file.
- In the new window
  - Indicate the place where mingw and g95 binaries are located
  - In `language for compilation` chose C/C++ or g95 depending on if the MEX-file will be created from a C or a FORTRAN subroutine.
  - Press `Make options file`

Now we should be able to create the MEX-file using the command `mex` in MATLAB.

A simple example of how to create a MEX-file is described in Sect. 2.7.1.

2.7.1 A Simple Example: Matrix Multiplication

In order to illustrate the procedure for constructing a MEX-file and how such MEX-file can speed-up our code, we consider in this section the simple example of matrix multiplication.

We first list the MATLAB algorithm for matrix multiplication `matrix_mult.m`

```
function C = matrix_mult(A,B,nrA,ncA,ncB)

% Code for matrix multiplication
C = zeros(nrA,ncB);
for ii = 1:nrA
    for jj = 1:ncB
        for kk = 1:ncA
            C(ii,jj) = C(ii,jj) + A(ii,kk)*B(kk,jj);
        end
    end
end
```

Function matrix_mult.m Simple MATLAB algorithm for matrix multiplication

It should be noted that the MATLAB symbol * contains a much more efficient algorithm than `matrix_mult.m` thus `matrix_mult.m` is only used for illustration purposes.

The FORTRAN code for matrix multiplication is written in `matmult.f`

```
C matmult subroutine in FORTRAN

     SUBROUTINE matmult(A,B,C,nrA,ncA,ncB)

     IMPLICIT NONE
     INTEGER nrA,ncA,ncB
     REAL*8 A(nrA,ncA)
     REAL*8 B(ncA,ncB)
     REAL*8 C(nrA,ncB)

     INTEGER I,J,K

     DO I=1,nrA
         DO J=1,ncB
             C(I,J) = 0.0
                 DO K=1,ncA
                     C(I,J) = C(I,J) + A(I,K)*B(K,J)
                 END DO
         END DO
     END DO
     RETURN
     END
```

Function matmult.f Simple FORTRAN algorithm for matrix multiplication

The next step is to create the gateway for the FORTRAN subroutine. For that purpose, we will employ the tool `OpenFGG`. In this tool, we must define the input and output variables of the subroutine, so we define `A`, `B`, `nrA`, `ncA`, and `ncB` as input variables and `C` as the output variable. After this step the gateway `matmultgw.f` is generated.[1] It must be noted here that the gateway `matmultgw.f` contains almost 700 lines of code which gives us an idea of the complexity of creating FORTRAN gateways by hand.

The next step is to create the MEX-file using the following command in MATLAB:

```
mex -O matmult.f matmultgw.f
```

where the `-O` option is to optimize the code. This step creates the MEX-file whose extension will depend on the platform and the version of MATLAB. In our case, we obtain `matmult.mexglx` since we are runing MATLAB 2009b under a Linux 32 bits platform.

[1] The code for the gateway `matmultgw.f` is not included in this document because of its length.

Another option is to create the MEX file for a C function. The C code for matrix multiplication is written in `mmcsubroutine.c`

```
void mmcsubroutine(
                   double    C[],
                   double    A[],
                   double    B[],
                   int       nrA,
                   int       ncA,
                   int       ncB
            )
{
    int i,j,k,cont;
    cont = 0;
    for (j=0; j<ncB; j++) {
        for (i=0; i<nrA; i++) {
            for (k=0; k<ncA; k++) {
                C[cont] += A[nrA*k+i]*B[k+j*ncA];
            }
            cont++;
        }
    }
    return;
}
```

Function mmcsubroutine.c Simple C algorithm for matrix multiplication

There is no software for the automatic generation of the gateway thus we have to create it by hand. In this particular case, this task is not too difficult but some programming skills are required. The gateway is written in `matmultc.c`

```
/* Gateway MATMULTC.C for matrix multiplication in C  */
#include <math.h>
#include "mex.h"

/* Input Arguments */
#define A_IN    prhs[0]
#define B_IN    prhs[1]
#define NRA_IN  prhs[2]
#define NCA_IN  prhs[3]
#define NCB_IN  prhs[4]

/* Output Arguments */
#define C_OUT   plhs[0]

/* Mex function */
void mexFunction( int nlhs, mxArray *plhs[],
          int nrhs, const mxArray*prhs[] )
{
    double *C;
    double *A,*B;
    int nrA,ncA,ncB;
    mwSize m,n,nA,mB;

    /* Check for proper number of arguments */
```

```
    if (nrhs != 5) {
    mexErrMsgTxt("Five input arguments required.");
    } else if (nlhs > 1) {
    mexErrMsgTxt("Too many output arguments.");
    }

    /* Check the dimensions of A and B */
    nA = mxGetN(A_IN);
    mB = mxGetM(B_IN);
    if ((nA != mB)) {
    mexErrMsgTxt("N cols. in A different from N rows in B.");
    }

    /* Create a matrix for the return argument */
    m = mxGetM(A_IN);
    n = mxGetN(B_IN);
    C_OUT = mxCreateDoubleMatrix(m, n, mxREAL);

    /* Assign pointers to the various parameters */
    C   = mxGetPr(C_OUT);
    A   = mxGetPr(A_IN);
    B   = mxGetPr(B_IN);
    nrA = (int) mxGetScalar(NRA_IN);
    ncA = (int) mxGetScalar(NCA_IN);
    ncB = (int) mxGetScalar(NCB_IN);

    /* Do the actual computations in a subroutine */
    mmcsubroutine(C,A,B,nrA,ncA,ncB);
    return;
}
```

Function matmultc.c C gateway for the subroutine `mmcsubroutine.c`

We can note the following details about `matmultc.c`:

1. The code starts by defining the input and output arguments of the subroutine `mmcsubroutine.c`.

```
/* Input Arguments */
#define A_IN    prhs[0]
#define B_IN    prhs[1]
#define NRA_IN  prhs[2]
#define NCA_IN  prhs[3]
#define NCB_IN  prhs[4]
/* Output Arguments */
#define C_OUT   plhs[0]
```

   There are five inputs and one output. The part `rhs` in `prhs[*]` calls for the right hand side while `lhs` in `plhs[*]` calls for the left hand side. The order of the parameters when calling the subroutine is indicated by the number between brackets.
2. After defining the inputs and outputs, we start by constructing the MEX function.

```
/* Mex function */
void mexFunction( int nlhs, mxArray *plhs[],
          int nrhs, const mxArray*prhs[] )
```

The name of the MEX function must always be `mexFunction`. This part is common to all MEX-files.

3. Next the parameters used in the MEX function are defined

```
double *C;
double *A,*B;
int nrA,ncA,ncB;
mwSize m,n,nA,mB;
```

4. The matrix output is created

```
/* Create a matrix for the return argument */
 m = mxGetM(A_IN);
 n = mxGetN(B_IN);
 C_OUT = mxCreateDoubleMatrix(m, n, mxREAL);
```

5. Then the pointers are assigned to the different input/output parameters

```
/* Assign pointers to the various parameters */
C   = mxGetPr(C_OUT);
A   = mxGetPr(A_IN);
B   = mxGetPr(B_IN);
nrA = (int) mxGetScalar(NRA_IN);
ncA = (int) mxGetScalar(NCA_IN);
ncB = (int) mxGetScalar(NCB_IN);
```

6. At this point, the subroutine which performs the matrix multiplication computation is called

```
/* Do the actual computations in a subroutine */
mmcsubroutine(C,A,B,nrA,ncA,ncB);
```

7. Alternatively some code for checking the input/output arguments can be included

```
/* Check for proper number of arguments */
    if (nrhs != 5) {
    mexErrMsgTxt(''Five input arguments required.'');
    } else if (nlhs > 1){
    mexErrMsgTxt(''Too many output arguments.'');
    }
    /* Check the dimensions of A and B */
    nA = mxGetN(A_IN); mB = mxGetM(B_IN);
    if ((nA != mB)) {
mexErrMsgTxt(''N cols. in A different from N rows in B.'');
    }
```

Finally, the MEX-file is created as in the previous case:

```
mex -O matmultc.c mmcsubroutine.c
```

Now we are ready to call the mex-file from MATLAB. In order to compare the computational times obtained with the MATLAB function and with the MEX-file, the MATLAB script `main_matrix_mult` will be used.

```
% Main program calling the matrix_mult code and the mex file
clear all
clc

% Dimension of matrices
nrA = 200;
ncA = 1000;
ncB = 500;

% Create the matrices
A = rand(nrA,ncA);
B = rand(ncA,ncB);

% Call the MATLAB function
tt = cputime;
C1 = matrix_mult(A,B,nrA,ncA,ncB);
fprintf('Matlab function time  = %2.2f s\n',cputime-tt);


% Call the FORTRAN MEX-file
tt = cputime;
C2 = matmult(A,B,nrA,ncA,ncB);
fprintf('Fortran Mex-file time = %2.2f s\n',cputime-tt);

% Call the C MEX-file
tt = cputime;
C3 = matmultc(A,B,nrA,ncA,ncB);
fprintf('C Mex-file time       = %2.2f s\n',cputime-tt);
```

Script main_matrix_mult Main program that calls MATLAB function `matrix_mult.m` and mex-file `matmult.mexglx`

As a result, we obtain the following MATLAB screen-print:

```
MATLAB function time  = 3.60 s
Fortran Mex-file time = 0.56 s
C Mex-file time       = 0.45 s
```

showing that for this particular case, the MEX-file obtained from the FORTRAN code is more than six times faster than the MATLAB function while the MEX-file obtained from the C code is even faster.

2.7.2 MEX-Files for ODE Solvers

Efficient FORTRAN or C time integrators can also be exploited within MATLAB using the concept of MEX-files. The creation of these MEX-files is too complex to be detailed in this introductory text and we content ourselves with an application example. A comparison against the IVP solver `RKF45` is shown in Table 2.14, which lists computational times normalized with respect to the smallest cost, i.e., that corresponding to the simulation of the logisticequation using the FORTRAN version.

Table 2.14 Computation times required for solving different ODE systems considered in this chapter with versions of the RKF45 solver implemented in FORTRAN and MATLAB

	FORTRAN	MATLAB
Logistic equation (Sect. 2.1)	1	1,500
Stiff ODEs (Sect. 2.2)	14.71	5882.4
Spruce budworm (Sect. 2.5)	229.4	56,971

As shown in the table, the FORTRAN MEX-file is orders of magnitude faster than the MATLAB version. Several MEX-files are available in the companion software. The details of their construction are omitted due to space limitation.

2.8 Summary

In this chapter, a few time integrators are detailed and coded so as to show the main ingredients of a good ODE solver. First, fixed-step integrators are introduced, followed by variable-step integrators that allow to achieve a prescribed level of accuracy. However, stability appears as an even more important issue than accuracy, limiting the time-step size in problems involving different time scales. A Rosenbrock's method is then considered as an example to solve efficiently this class of problems. After the presentation of these several basic time integrators, we turn our attention to the MATLAB ODE suite, a powerful library of time integrators which allow to solve a wide range of problems. We then apply several of these integrators to two more challenging application examples, i.e., the study of spruce budworm dynamics and the study of liming to remediate the effect of acid rain on a lake. On the other hand, we introduce the use of SCILAB and OCTAVE, two attractive open-source alternatives to MATLAB, to solve ODE and DAE problems, and we highlight the main syntaxic differences. As MATLAB is an interpreted language, the computational cost can however be an issue when computing the solution of large system of ODEs, or when solving repeatedly the same problem (as for instance when optimizing a cost function, involving the solution of a ODE model). The use of compiled functions can be advantageous in this case, and this is why we end the presentation of ODE integrators by the use of MATLAB MEX-files.

References

1. Shampine LF, Gladwell I, Thompson S (2003) Solving ODEs with MATLAB. Cambridge University Press, Cambridge
2. Hindmarsh AC (1983) ODEPACK, a systematized collection of ODE solvers, in scientific computing. In: Stepleman RS (ed) IMACS transactions on scientific computation, vol. 1. Elsevier, Amsterdam

3. Hindmarsh AC (1980) Lsode and lsodi, two new initial value ordinary differential equation solvers. ACM Signum Newslett 15(4):10–11
4. Breman KE, Campbell SL, Petzold LR (1989) Numerical solution of initial-value problems in differential-algebraic equations. Elsevier, New York
5. Verhulst PF (1938) Notice sur la loi que la population poursuit dans son accroissement. Correspondance mathmatique et physique 10:113–121
6. Butcher JC (1987) The numerical analysis of ordinary differential equations. Wiley, New York
7. Lambert JD (1991) Numerical methods for ordinary differential systems. Wiley, London
8. Hairer E, Wanner G (1996) Solving ordinary differential equations II: stiff and differential algebraic problems. Springer, New York
9. Hairer E, Norsett SP, Wanner G (1993) Solving ordinary differential equations I: nonstiff problems. Springer, New York
10. Rosenbrock HH (1963) Some general implicit processes for the numerical solution of differential equations. Comput J 5:329–330
11. Verwer JG, Spee EJ, Blom JG, Hundsdorfer W (1999) A second order rosenbrock method applied to photochemical dispersion problems. J Sci Comput 20:1456–1480
12. Lang J, Verwer JG (2001) Ros3p—an accurate third-order rosenbrock solver designed for parabolic problems. BIT 21:731–738
13. Ludwig D, Jones DD, Holling CS (1995) Qualitative analysis of insect outbreak systems: the spruce budworm and forest. J Anim Ecol 47:315–332
14. Ghosh M (2002) Effect of liming on a fish population in an acidified lake: a simple mathematical model. Appl Math Comput 135(2–3):553–560
15. Gomez C (2003) SCILAB consortium launched. Technical report 54, ERCIM News
16. Eaton JW, Bateman D, Hauberg S, Wehbring R (2013) GNU Octave: a high-level interactive language for numerical computations, 3rd edn. Network Theory Ltd, Bristol

http://www.springer.com/978-3-319-06789-6

Simulation of ODE/PDE Models with MATLAB®, OCTAVE and SCILAB
Scientific and Engineering Applications
Vande Wouwer, A.; Saucez, P.; Vilas Fernández, C.
2014, XV, 406 p. 141 illus., 33 illus. in color. With online files/update., Hardcover
ISBN: 978-3-319-06789-6