CHAPTER 1

Introduction to Ordinary Differential Equation Analysis: Bioreactor Dynamics

1.1 Introduction

Mathematical models formulated as systems of ordinary differential equations (ODEs) and partial differential equations (PDEs) have been reported for a spectrum of applications in biomedical science and engineering (BMSE). The intent of this research is to provide a quantitative understanding of the biological, chemical, and physical phenomena that determine the characteristics of BMSE systems and to provide a framework for the analysis and interpretation of experimental data observed in the study of BMSE systems.

In the subsequent discussion in this chapter, we consider the programming of a 7×7 (seven equations in seven unknowns) ODE system to illustrate the integration (solution) of ODE systems using R, a quality, open source, scientific programming system [10]. The intent is to provide the reader with a complete and thoroughly documented example of the numerical integration of an ODE system, including (i) the use of library ODE integrators, (ii) the programming of ODE integration algorithms, and (iii) graphical output of the numerical solutions. This example application can then serve as a prototype or template which the reader can modify and extend for an ODE model of interest.

Differential Equation Analysis in Biomedical Science and Engineering: Ordinary Differential Equation Applications with R, First Edition. William E. Schiesser.

1.2 A 7 × 7 ODE System for a Bioreactor

The reaction system for the conversion of xylose to ethanol by fermentation is now formulated and coded (programmed) in R. The ODE model is discussed in detail in [1, pp 35–42]; this discussion is recommended as a starting point for the details of the chemical reactions, particularly the various intermediates, so that the discussion to follow can concentrate on the numerical algorithms and R programming.

The reaction system is given in Table 1.1.

TABLE 1.1 Summary of reactions.[a]

Reaction Number	Reaction Stoichiometry
1	xylose $\rightleftharpoons$ xylitol
2	xylitol $\rightleftharpoons$ xylulose
3	2 xylulose $\rightleftharpoons$ 3 acetaldehyde
4	acetaldehyde $\rightleftharpoons$ ethanol
5	acetaldehyde $\rightleftharpoons$ acetate
6	2 xylulose $\rightleftharpoons$ 3 glycerol

[a] From [1], Table 2.1, p 39.

The corresponding ODE system is [1, p 39]

$$\frac{d[\text{xylose}]}{dt} = -J_1 \tag{1.1a}$$

$$\frac{d[\text{xylitol}]}{dt} = J_1 - J_2 \tag{1.1b}$$

$$\frac{d[\text{xylulose}]}{dt} = J_2 - 2J_3 - 2J_6 \tag{1.1c}$$

$$\frac{d[\text{acetaldehyde}]}{dt} = 3J_3 - J_4 - J_5 \tag{1.1d}$$

$$\frac{d[\text{ethanol}]}{dt} = J_4 \tag{1.1e}$$

$$\frac{d[\text{acetate}]}{dt} = J_5 \tag{1.1f}$$

$$\frac{d[\text{glycerol}]}{dt} = 3J_6 \tag{1.1g}$$

The concentrations in eqs. (1.1), denoted as [ ], are expressed in total (intraplus extracellular) moles per unit cell dry weight.

J_1 to J_6 are the kinetic rates for the six reactions listed in Table 1.1. The multiplying constants are stoichiometric coefficients. For example, reaction 3 (with rate J_3) in Table 1.1 produces 3 mol of acetaldehyde for every 2 mol of xylulose consumed. Therefore, eq. (1.1c) for $d[\text{xylulose}]/dt$ has -2 multiplying J_3 and eq. (1.1d) for $d[\text{acetaldehyde}]/dt$ has $+3$ multiplying J_3.

The reaction rates, J_1 to J_6, are expressed through mass action kinetics.

$$J_1 = k_1[\text{xylose}] \tag{1.2a}$$

$$J_2 = k_2[\text{xylitol}] - k_{-2}[\text{xylulose}][\text{ethanol}] \tag{1.2b}$$

$$J_3 = k_3[\text{xylulose}] - k_{-3}[\text{acetaldehyde}][\text{ethanol}] \tag{1.2c}$$

$$J_4 = k_4[\text{acetaldehyde}] \tag{1.2d}$$

$$J_5 = k_5[\text{acetaldehyde}] \tag{1.2e}$$

$$J_6 = k_6[\text{xylulose}] \tag{1.2f}$$

Note in particular the product terms for the reverse reactions in eqs. (1.2b) and (1.2c), $-k_{-2}[\text{xylulose}][\text{ethanol}]$ and $-k_{-3}[\text{acetaldehyde}][\text{ethanol}]$, which are nonlinear and therefore make the associated ODEs nonlinear (with right-hand side (RHS) terms in eqs. (1.1) that include J_2 and J_3). This nonlinearity precludes the usual procedures for the analytical solution of ODEs based on the linear algebra, that is, a numerical procedure is required for the solution of eqs. (1.1).

k_1 to k_6, k_{-2}, k_{-3}, in eqs. (1.2) are kinetic constants (adjustable parameters) that are selected so that the model output matches experimental data in some manner, for example, a least squares sense. Two sets of numerical values are listed in Table 1.2

`BP000` refers to a wild-type yeast strain, while `BP10001` refers to an engineered yeast strain.

TABLE 1.2 Kinetic constants for two yeast strains.[a]

Parameter	Value (BP000)	Value (BP10001)	Units
k_1	7.67×10^{-3}	8.87×10^{-3}	h^{-1}
k_2	3.60	13.18	h^{-1}
k_3	0.065	0.129	h^{-1}
k_4	0.867	0.497	h^{-1}
k_5	0.045	0.027	h^{-1}
k_6	1.15×10^{-3}	0.545×10^{-3}	h^{-1}
k_{-2}	88.0	88.7	$gh^{-1}\ mol^{-1}$
k_{-3}	99.0	99.9	$gh^{-1}\ mol^{-1}$

[a] From [1], Table 2.2, p 41.

To complete the specification of the ODE system, each of eqs. (1.1) requires an initial condition (IC) (and only one IC because these equations are first order in t).

TABLE 1.3 Initial conditions (ICs) for eqs. (1.1).

Equation	IC (t=0)
(1.1a)	[xylose] = 0.10724
(1.1b)	[xylitol] = 0
(1.1c)	[xylulose] = 0
(1.1d)	[acetaldehyde] = 0
(1.1e)	[ethanol] = 0
(1.1f)	[acetate] = 0
(1.1g)	[glycerol] = 0

The 7×7 ODE system is now completely defined and we can proceed to programming the numerical solution.

1.3 In-Line ODE Routine

An ODE routine for eqs. (1.1) is listed in the following.

```
#
# Library of R ODE solvers
```

```
  library("deSolve")
#
# Parameter values for BP10001
  k1=8.87e-03;
  k2=13.18;
  k3=0.129;
  k4=0.497;
  k5=0.027;
  k6=0.545e-3;
  km2=87.7;
  km3=99.9;
#
# Initial condition
  yini=c(y1=0.10724,y2=0,y3=0,y4=0,y5=0,y6=0,y7=0)
  yini
  ncall=0;
#
# t interval
  nout=51
  times=seq(from=0,to=2000,by=40)
#
# ODE programming
  bioreactor_1=function(t,y,parms) {
  with(as.list(y),
    {
#
# Assign state variables:
  xylose      =y1;
  xylitol     =y2;
  xylulose    =y3;
  acetaldehyde=y4;
  ethanol     =y5;
  acetate     =y6;
  glycerol    =y7;
#
# Fluxes
  J1=k1*xylose;
  J2=k2*xylitol-km2*xylulose*ethanol;
  J3=k3*xylulose-km3*acetaldehyde*ethanol;
  J4=k4*acetaldehyde;
  J5=k5*acetaldehyde;
```

```
  J6=k6*xylulose;
#
# Time derivatives
  f1=-J1;
  f2=J1-J2;
  f3=J2-2*J3-2*J6;
  f4=3*J3-J4-J5;
  f5=J4;
  f6=J5;
  f7=3*J6;
#
# Calls to bioreactor_1
  ncall <<- ncall+1
#
# Return derivative vector
  list(c(f1,f2,f3,f4,f5,f6,f7))
  })
}
#
# ODE integration
  out=ode(y=yini,times=times,func=bioreactor_1,parms=NULL)
#
# ODE numerical solution
  for(it in 1:nout){
   if(it==1){
   cat(sprintf(
   "\n        t      y1      y2      y3      y4      y5
      y6      y7"))}
   cat(sprintf("\n %8.0f%8.4f%8.4f%8.4f%8.4f%8.4f%8.4f
      %8.4f",
   out[it,1],out[it,2],out[it,3],out[it,4],
   out[it,5],out[it,6],out[it,7],out[it,8]))
  }
#
# Calls to bioreactor_1
  cat(sprintf("\n ncall = %5d\n\n",ncall))
#
# Set of 7 plots
  plot(out)
```

Listing 1.1: ODE routine.

We can note the following details about Listing 1.1.

- The R library of ODE numerical integrators, `deSolve`, is specified. The contents of this library will be discussed subsequently through examples.

```
#
# Library of R ODE solvers
  library("deSolve")
```

- The parameters from Table 1.2 for the engineered yeast strain `BP10001` are defined numerically.

```
#
# Parameter values for BP10001
  k1=8.87e-03;
  k2=13.18;
  k3=0.129;
  k4=0.497;
  k5=0.027;
  k6=0.545e-3;
  km2=87.7;
  km3=99.9;
```

- The ICs of Table 1.3 are defined numerically through the use of the R vector utility `c` (which defines a vector, in this case `yini`). This statement illustrates a feature of R that requires careful attention, that is, there are reserved names such as `c` that should not be used in other ways such as the definition of a variable with the name `c`.

```
#
# Initial condition
  yini=c(y1=0.10724,y2=0,y3=0,y4=0,y5=0,y6=0,y7=0)
  yini
  ncall=0;
```

  Also, the naming of the variables is open for choice (except for reserved names). Here, we select something easy to program, that is, `y1` to `y7` but programming in terms of problem-oriented variables is illustrated subsequently. Also, the elements in the

IC vector `yini` are displayed by listing the name of the vector on a separate line. This is an obvious but important step to ensure that the ICs are correct as a starting point for the solution. Finally, the number of calls to the ODE function, `bioreactor_1`, is initialized.

- The values of t (in eqs. (1.1)) at which the solution is to be displayed are defined as the vector `times`. In this case, the R function `seq` is used to define the sequence of 51 values $t = 0, 40, \ldots, 2000$.

```
#
# t interval
  nout=51
  times=seq(from=0,to=2000,by=40)
```

To give good resolution (smoothness) of the plots of the solutions, 51 was selected (discussed subsequently).

- Eqs. (1.1) are programmed in a function `bioreactor_1`.

```
#
# ODE programming
  bioreactor_1=function(t,y,parms) {
  with(as.list(y),
    {
```

We can note the following details about function `bioreactor_1`.

— The function is defined with three input arguments, `t,y,parms`. Also, a left brace, `{`, is used to start the function that is matched with a right brace, `}`, at the end of the function.

— The input argument `y` is a list (rather than a numerical vector) specified with `with(as.list(y)`, (this statement is optional and is not used in subsequent ODE routines). The second `{` starts the `with` statement.

— The seven dependent variables, `y1` to `y7`, are placed in problem-oriented variables, `xylose` to `glycerol`, to facilitate the programming of eqs. (1.1).

```
#
# Assign state variables:
  xylose      =y1;
  xylitol     =y2;
  xylulose    =y3;
  acetaldehyde=y4;
  ethanol     =y5;
  acetate     =y6;
  glycerol    =y7;
```

— The fluxes of eqs. (1.2) are programmed.

```
#
# Compute fluxes
  J1=k1*xylose;
  J2=k2*xylitol-km2*xylulose*ethanol;
  J3=k3*xylulose-km3*acetaldehyde*ethanol;
  J4=k4*acetaldehyde;
  J5=k5*acetaldehyde;
  J6=k6*xylulose;
```

— The ODEs of eqs. (1.1) are programmed, with the left-hand side (LHS) derivatives placed in the variables `f1` to `f7`. For example, d[xylose]/$dt \rightarrow$ `f1`.

```
#
# Time derivatives
  f1=-J1;
  f2=J1-J2;
  f3=J2-2*J3-2*J6;
  f4=3*J3-J4-J5;
  f5=J4;
  f6=J5;
  f7=3*J6;
```

— The number of calls to `bioreactor_1` is incremented and returned to the calling program with `<<-`.

```
#
# Calls to bioreactor_1
  ncall <<- ncall+1
```

This use of <<- illustrates a basic property of R, that is, numerical values set in a subordinate routine are not shared with higher level routines without explicit programming such as <<-.

— The vector of derivatives is returned from bioreactor_1 as a list.

```
#
# Return derivative vector
  list(c(f1,f2,f3,f4,f5,f6,f7))
  })
}
```

Note the use of the R vector utility c. The }) ends the with statement and the second } concludes the function bioreactor_1. In other words, the derivative vector is returned from bioreactor_1 as a list. This is a requirement of the ODE integrators in the library deSolve. This completes the programming of bioreactor_1. We should note that this function is part of the program of Listing 1.1. That is, this function is in-line and is defined (programmed) before it is called (used). An alternative would be to formulate bioreactor_1 as a separate function; this is done in the next example.

• Eqs. (1.1) are integrated numerically by a call to the R library integrator ode (which is part of deSolve).

```
#
# ODE integration
  out=ode(y=yini,times=times,func=bioreactor_1,
     parms=NULL)
```

We can note the following details about this call to ode.

— The inputs to ode are (i) yini, the IC vector; (ii) times, the vector of output values of t; and (iii) bioreactor_1 to define the RHSs of eqs. (1.1). These inputs define the ODE system of eqs. (1.1) as expected.

— The fourth input argument, `parms`, can be used to provide a vector of parameters. In the present case, it is unused. However, a vector of parameters, `k1` to `km3`, was defined previously for use in `bioreactor_1`. This sharing of the parameters with `bioreactor_1` illustrates a basic property of R: Numerical values set in a higher level routine are shared with subordinate routines (e.g., functions) without any special designation for this sharing to occur.

— `ode` has as a default the ODE integrator `lsoda` [10]. The `a` in the name `lsoda` stands for "automatic," meaning that `lsoda` automatically switches between a stiff option and a nonstiff option as the numerical integration of the ODE system proceeds. The significance of stiffness will be discussed in the following and in subsequent chapters. Here we mention only that this is a sophisticated feature intended to relieve the analyst of having to specify a stiff or nonstiff integrator. `lsoda` also has a selection of options that can be specified when it is called via `ode` such as error tolerances for the ODE integration. Experimentation with these options (rather than the use of the defaults) may improve the performance of `ode`. In the present case, only the defaults are used.

— The numerical solution of the ODE system is returned from `ode` as a 2D array, in this case `out`. The first index of this solution array is for the output values of the independent variable (t). The second index is for the numerical solution of the ODEs. For example, `out` in the present case has the dimensions `out[51,1+7]` corresponding to (i) the 51 output values $t = 0, 40, \ldots, 2000$ (defined previously) and (ii) the seven dependent variables of eqs. (1.1) plus the one independent variable t. For example, `out[1,1]` is the value $t = 0$ and `out[51,1]` is the value $t = 2000$. `out[1,2]` is (from eq. (1.1a) and Table 1.3) [xylose]($t = 0$) = 0.10724 and `out[51,2]` is [xylose]($t = 2000$). `out[1,8]` is (from eq. (1.1g) and Table 1.3) [glycerol]($t = 0$) = 0 and `out[51,8]` is [glycerol]($t = 2000$). An understanding of the arrangement

of the output array is essential for subsequent numerical and graphical (plotted) display of the solution.

— `ode` receives the number of output values of the solution from the length of the vector of output values of the independent variable. For example, `times` has 51 elements ($t = 0, 40, \ldots, 2000$) that define the first dimension of the output array as 51 (in `out[51,1+7]`).

— `ode` receives the number of ODEs to be integrated from the length of the IC vector. For example, `yini` has seven elements that define the second dimension of the output array as `out[51,1+7]` (with the one added to include t).

- The numerical solution is displayed at the `nout` =51 output values of t through a `for` loop. For `it=1` ($t = 0$), a heading for the numerical solution is displayed.

```
#
# ODE numerical solution
  for(it in 1:nout){
   if(it==1){
   cat(sprintf(
   "\n        t       y1       y2       y3       y4
      y5       y6       y7"))}
   cat(sprintf("\n %8.0f%8.4f%8.4f%8.4f%8.4f%8.4f%8.4f
      %8.4f",
   out[it,1],out[it,2],out[it,3],out[it,4],
   out[it,5],out[it,6],out[it,7],out[it,8]))
  }
```

Note the use of the $51 \times (1 + 7)$ values in `out`. Also, the combination of the R utilities `cat` and `sprintf` provides formatting that is used in other languages (e.g., C, C++, Matlab).

- The number of calls to `bioreactor_1` is displayed at the end of the solution to give an indication of the computational effort required to compute the solution.

```
#
# Calls to bioreactor_1
  cat(sprintf("\n ncall = %5d\n\n",ncall))
```

- Finally, the solutions of eqs. (1.1) are plotted with the R utility `plot`.

```
#
# Set of 7 plots
  plot(out)
```

A complete plot is produced with just this abbreviated use of `out`. `plot` has a variety of options to format the graphical output that will be considered in subsequent applications.

1.4 Numerical and Graphical Outputs

Abbreviated numerical output from Listing 1.1 is given in Table 1.4.

We can note the following details about this output.

TABLE 1.4 Abbreviated numerical output from Listing 1.1.

t	y1	y2	y3	y4	y5	y6	y7
0	0.1072	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
40	0.0752	0.0020	0.0153	0.0009	0.0195	0.0011	0.0006
80	0.0527	0.0053	0.0221	0.0008	0.0361	0.0020	0.0018
120	0.0370	0.0081	0.0245	0.0006	0.0497	0.0027	0.0034
160	0.0259	0.0101	0.0248	0.0005	0.0608	0.0033	0.0050
200	0.0182	0.0112	0.0239	0.0004	0.0702	0.0038	0.0066
		.				.	
		.				.	
		.				.	
		Output for t = 240 to 1760 removed					
		.				.	
		.				.	
		.				.	
1800	0.0000	0.0004	0.0004	0.0000	0.1303	0.0071	0.0222
1840	0.0000	0.0003	0.0004	0.0000	0.1304	0.0071	0.0223
1880	0.0000	0.0003	0.0004	0.0000	0.1305	0.0071	0.0223
1920	0.0000	0.0003	0.0003	0.0000	0.1306	0.0071	0.0223
1960	0.0000	0.0003	0.0003	0.0000	0.1306	0.0071	0.0223
2000	0.0000	0.0002	0.0003	0.0000	0.1307	0.0071	0.0223

```
ncall =   427
```

- The ICs (at $t = 0$) correspond to the values in Table 1.3. While this may seem to be an obvious fact, it is a worthwhile check to ensure that the solution has the correct starting values.
- The solutions approach steady-state conditions as $t \to 2000$. Note in particular that `y1` (for xylose from eq. (1.1a)) approaches zero as the the reactant that drives the system is nearly consumed. Also, `y5` (for ethanol from eq. (1.1e)) approaches 0.1307 indicating a significant production of ethanol, the product of primary interest (e.g., possibly to be used as a fuel). `y7` (for glycerol from eq. (1.1g)) approaches 0.0223 and might represent a contaminant that would have to be subsequently reduced by a separation process; this is rather typical of reaction systems, that is, they usually produce undesirable by-products.
- The computational effort is quite modest, `ncall = 427` (the reason for calling this "modest" is explained subsequently).

The graphical output is given in Fig. 1.1. We can note the following about Fig. 1.1.

- The plotting utility `plot` provides automatic scaling of each of the seven dependent variables. Also, the default of `plot` is the solid lines connecting the values in Table 1.4; alternative options provide discrete points, or points connected by lines.
- The initial ($t = 0$) values reflect the ICs of Table 1.3 and the final values ($t \to 2000$) reflect the values of Table 1.4.
- The solutions have their largest derivatives at the beginning which is typical of ODE systems (the LHSs of eqs. (1.2) is largest initially).
- The plots are smooth with 51 points.

A fundamental question remains concerning the accuracy of the solution in Table 1.4. As an exact (i.e., analytical, mathematical, closed form) solution is not available for eqs. (1.1) (primarily because they are nonlinear as discussed previously), we cannot directly determine the accuracy of the numerical solution by comparison with an

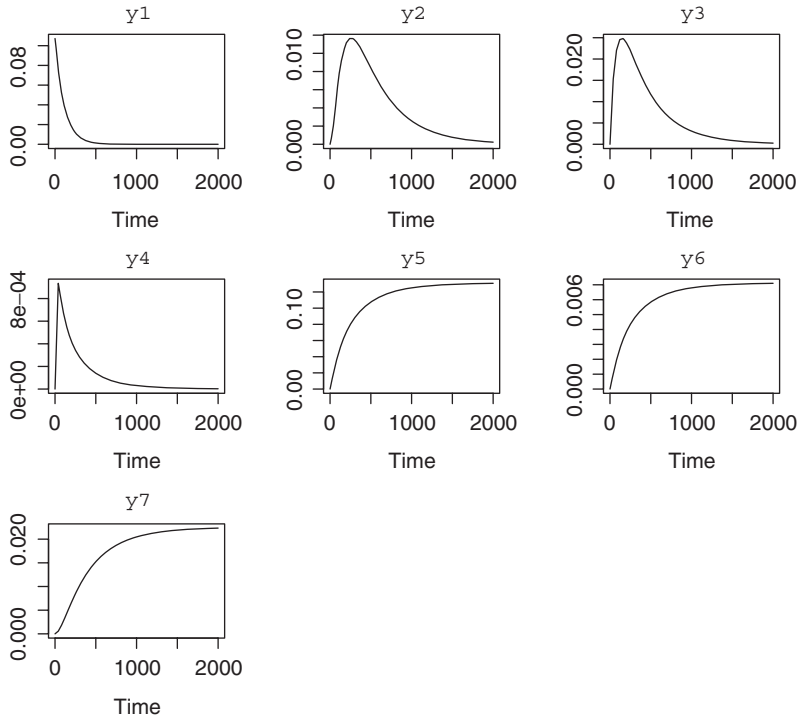

Figure 1.1 Solutions to eqs. (1.1).

exact solution (and if such a solution was available, there would really be no need to compute a numerical solution).

We therefore must use a method of accuracy evaluation that is built on the numerical approach. For example, we could change the specified error tolerances for `lsoda` (via the call to `ode`) and compare the solutions as the error tolerances are changed. Or we could use other ODE integrators (other than `lsoda`) and compare the solutions from different integrators (this approach is discussed in a subsequent example). In any case, some form of error analysis is an essential part of any numerical procedure to give reasonable confidence that the numerical solution has acceptable accuracy.

Finally, with the operational code of Listing 1.1, we can now perform studies (experiments) that will contribute to an understanding of the problem system (which is usually the ultimate objective in developing a mathematical model) on the computer. For example,

the effect of changing the model parameters, termed the *parameter sensitivity*, can be carried out by observing the changes in the solutions as parameters are varied. As an example, the `BP000` parameters of Table 1.2 can be used in place of the `BP10001` parameters (in Listing 1.1) to investigate the effect of using an engineered yeast strain (`BP10001`) in place of a wild-type yeast strain (`BP000`). Ideally, an increase in ethanol production would be observed (the final value of `y5` in Table 1.4 would increase), indicating that the engineered yeast strain can improve the efficiency of ethanol production.

This type of parameter sensitivity analysis presupposes available values of the model parameters that reflect the performance of the problem system, and these parameters might have to be measured experimentally, for example, by comparing the model solution with laboratory data, and/or estimated using available theory. A good example of the comparison of the ethanol model solution with experimental data is given in [1] for BP000 (Fig. 2.7) and BP10001 (Fig. 2.8).

1.5 Separate ODE Routine

Variations of the coding in Listing 1.1 will now be considered. The intent is to produce a more flexible modular format and to enhance the graphical output. The main program now is in Listing 1.2 (in place of Listing 1.1)

```
#
# Library of R ODE solvers
  library("deSolve")
#
# ODE routine
  setwd("c:/R/bme_ode/chap1")
  source("bioreactor_2.R")
#
# Parameter values for BP10001
  k1=8.87e-03;
  k2=13.18;
  k3=0.129;
  k4=0.497;
```

```
  k5=0.027;
  k6=0.545e-3;
  km2=87.7;
  km3=99.9;
#
# Initial condition
  yini=c(0.10724,0,0,0,0,0,0)
  ncall=0;
#
# t interval
  nout=51
  times=seq(from=0,to=2000,by=40)
#
# ODE integration
  out=ode(y=yini,times=times,func=bioreactor_2,parms=NULL)
#
# ODE numerical solution
  for(it in 1:nout){
   if(it==1){
   cat(sprintf(
   "\n        t      y1      y2      y3      y4      y5
      y6      y7"))}
   cat(sprintf("\n %8.0f%8.4f%8.4f%8.4f%8.4f%8.4f%8.4f
      %8.4f",
   out[it,1],out[it,2],out[it,3],out[it,4],
   out[it,5],out[it,6],out[it,7],out[it,8]))
  }
#
# Calls to bioreactor_2
  cat(sprintf("\n ncall = %5d\n\n",ncall))
#
# Single plot
  par(mfrow=c(1,1))
#
# y1
  plot(out[,1],out[,2],type="l",xlab="t",ylab="y1(t),...,
     y7(t)",
    xlim=c(0,2000),ylim=c(0,0.14),lty=1, main="y1(t),...,
       y7(t) vs t",
    lwd=2)
#
```

```
# y2
  lines(out[,1],out[,3],type="l",lty=2,lwd=2)
#
# y3
  lines(out[,1],out[,4],type="l",lty=3,lwd=2)
#
# y4
  lines(out[,1],out[,5],type="l",lty=4,lwd=2)
#
# y5
  lines(out[,1],out[,6],type="l",lty=5,lwd=2)
#
# y6
  lines(out[,1],out[,7],type="l",lty=6,lwd=2)
#
# y7
  lines(out[,1],out[,8],type="l",lty=7,lwd=2)
```

Listing 1.2: Main program with separate ODE routine.

We can note the following details about Listing 1.2.

- `library("deSolve")` is used again (as in Listing 1.1) in order to access the ODE integrator `ode`. In addition, the separate ODE routine `bioreactor_2` is accessed through the `setwd` and `source` R utilities.

  ```
  #
  # Library of R ODE solvers
    library("deSolve")
  #
  # ODE routine
    setwd("c:/R/bme_ode/chap1")
    source("bioreactor_2.R")
  ```

  To explain the use of `setwd` and `source`:

  — `setwd`, set woking directory, is used to go to a directory (folder) where the R routines are located. Note in particular the use of the forward slash `/` rather than the usual backslash \.

— `source` identifies a particular file within the directory identified by the `setwd`; in this case, the ODE routine `bioreactor_2` is called by `ode`.

— These two statements could be combined as

```
source("c:/R/bme_ode/chap1/bioreactor_2.R")
```

If the R application uses a series of files from the same directory, using the `setwd` is usually simpler; a series of `source` statements can then be used access the required files.

- The sections of Listing 1.2 for setting the parameters, IC and t interval, are the same as in Listing 1.1 and are therefore not discussed here. The call to `ode` uses the ODE routine `bioreactor_2` (Listing 1.3) rather than `bioreactor_1` (Listing 1.1).

```
#
# ODE integration
  out=ode(y=yini,times=times,func=bioreactor_2,
     parms=NULL)
```

Again, the ODE solution is returned in 2D array `out` for subsequent display. `bioreactor_2` is in a separate routine rather than placed in-line as in Listing 1.1, which makes the coding more modular and easier to follow.

- The display of the numerical solution is the same as in Listing 1.1 so this code is not discussed here.
- The number of calls to `bioreactor_3` (returned from `bioreactor_2` at the end of the solution, i.e., at $t = 2000$) is displayed.

```
#
# Calls to bioreactor_2
  cat(sprintf("\n ncall = %5d\n\n",ncall))
```

- The graphical output is extended to produce a single plot with the seven ODE solution curves.

```
#
# Single plot
```

```
  par(mfrow=c(1,1))
#
# y1
  plot(out[,1],out[,2],type="l",xlab="t",ylab="y1(t),
     ...,y7(t)",
    xlim=c(0,2000),ylim=c(0,0.14),lty=1, main="y1(t),
       ...,y7(t) vs t",
    lwd=2)
#
# y2
  lines(out[,1],out[,3],type="l",lty=2,lwd=2)
#
# y3
  lines(out[,1],out[,4],type="l",lty=3,lwd=2)
#
# y4
  lines(out[,1],out[,5],type="l",lty=4,lwd=2)
#
# y5
  lines(out[,1],out[,6],type="l",lty=5,lwd=2)
#
# y6
  lines(out[,1],out[,7],type="l",lty=6,lwd=2)
#
# y7
  lines(out[,1],out[,8],type="l",lty=7,lwd=2)
```

To explain this coding,

— A 1×1 array of plots is specified, that is, a single plot;

```
#
# Single plot
  par(mfrow=c(1,1))
```

— `plot` is used with a series of parameters for $y_1(t)$.

```
#
# y1
  plot(out[,1],out[,2],type="l",xlab="t",ylab="y1
     (t),...,y7(t)",
```

```
xlim=c(0,2000),ylim=c(0,0.14),lty=1, main="y1
   (t),...,y7(t) vs t",
lwd=2)
```

These parameters are:

`out[,1],out[,2]` plotted to give a solution curve for eq. (1.1a) of y_1 versus t;

`type="l"` designates a line type of the solution curve (rather than a point type);

`xlab="t"` specifies the label `t` on the abcissa (horizontal) axis;

`ylab="y1(t),...,y7(t)"` specifies the label on the ordinate (vertical) axis;

`xlim=c(0,2000)` scales the horizontal axis for $0 \leq t \leq 2000$;

`ylim=c(0,0.14)` scales the vertical axis to include the range of values from y_1 to y_7;

`lty=1` sets the type of line for the first solution as reflected in Fig. 1.2;

`main="y1(t),...,y7(t) vs t"` specifies a main label or title for the plot as reflected in Fig. 1.2;

`lwd=2` sets the line width for the first solution as reflected in Fig. 1.2.

- $y_2(t)$ is included as a second solution with the R utility `lines` by plotting `out[,1],out[,3]`. The parameters are the same as for the previous call to `plot` except `lty=2`, which specifies a second type of line as reflected in Fig. 1.2.

```
#
# y2
  lines(out[,1],out[,3],type="l",lty=2,lwd=2)
```

- $y_3(t)$ to $y_7(t)$ are plotted in the same way with `lines`. For example, $y_7(t)$ is plotted as

```
#
# y7
```

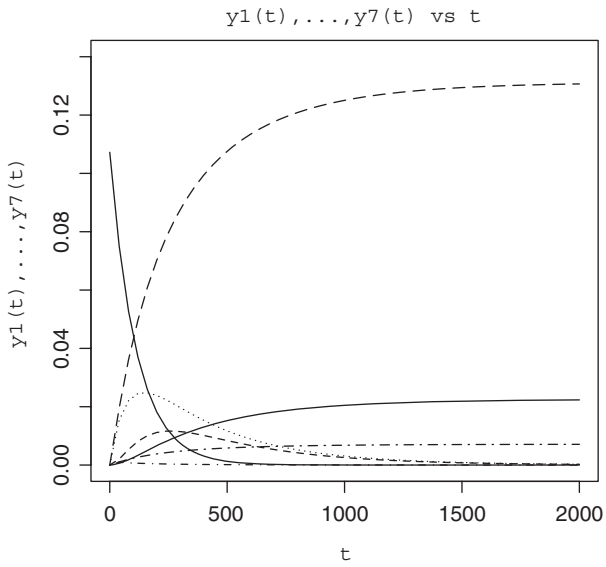

Figure 1.2 Solutions to eqs. (1.1) using a separate ODE routine.

```
lines(out[,1],out[,8],type="l",lty=7,lwd=2)
```

with a line type specified as `lty=7`, which specifies a seventh type of line as reflected in Fig. 1.2.

`bioreactor_2` in Listing 1.3 is a separate routine called by `ode`.

```
  bioreactor_2=function(t,y,parms) {
#
# Assign state variables:
  xylose      =y[1];
  xylitol     =y[2];
  xylulose    =y[3];
  acetaldehyde=y[4];
  ethanol     =y[5];
  acetate     =y[6];
  glycerol    =y[7];
#
# Compute fluxes
  J1=k1*xylose;
  J2=k2*xylitol-km2*xylulose*ethanol;
```

```
  J3=k3*xylulose-km3*acetaldehyde*ethanol;
  J4=k4*acetaldehyde;
  J5=k5*acetaldehyde;
  J6=k6*xylulose;
#
# Time derivatives
  f1=-J1;
  f2=J1-J2;
  f3=J2-2*J3-2*J6;
  f4=3*J3-J4-J5;
  f5=J4;
  f6=J5;
  f7=3*J6;
#
# Calls to bioreactor_2
  ncall <<- ncall+1
#
# Return derivative vector
  return(list(c(f1,f2,f3,f4,f5,f6,f7)))
}
```

Listing 1.3: ODE routine `bioreactor_2`.

`bioreactor_2` is the same as `bioreactor_1` of Listing 1.1 except for the following details.

- The function is defined as in Listing 1.1, but the statement specifying `y` as a list (`with(as.list(y))` is not used.

  ```
  bioreactor_2=function(t,y,parms) {
  ```

- The dependent variables constitute a vector (`y[1],...,y[7]`) rather than a list of scalars (`y1,...,y7`) as in Listing 1.1. In other words, the input argument of `bioreactor_2`, `y`, is a vector and not a list.
- At the end, the calls to `bioreactor_3` is incremented and returned to the main program of Listing 1.2 with `<<-`.

  ```
  #
  # Calls to bioreactor_2
    ncall <<- ncall+1
  ```

- Finally, the derivatives `f1` to `f7` are placed in a vector with `c` that is returned from `bioreactor_2` (to `lsoda` in `ode`) as a list (as required by the ODE integrators in `deSolve`).

```
#
# Return derivative vector
  return(list(c(f1,f2,f3,f4,f5,f6,f7)))
}
```

The right-hand bracket `}` concludes `bioreactor_2`.

The numerical output from Listing 1.2 is identical to that of Listing 1.1 as expected because the only difference in the coding is the use of the separate ODE routine `bioreactor_2`. The graphical output is given in Fig. 1.2. Note the composite plot of Fig 1.2 rather than the separate plots of Fig 1.1.

This concludes the example with a separate ODE routine. The intent is primarily to demonstrate the use of subordinate functions to modularize the code (rather than having it all in one routine such as in Listing 1.1). This modularization becomes increasingly useful as the complexity of the application increases because it can be used to organize the code into small, more easily manageable sections.

1.6 Alternative Forms of ODE Coding

We now consider variations of the coding in the preceding Listings 1.1–1.3. The intention is primarily to introduce alternatives that can be useful, particularly as the number of ODEs increases (beyond the 7 of eqs. (1.1)).

In the following example, the main program is the same as in Listing 1.2 except for the use of `bioreactor_3` in place of `bioreactor_2`.

```
#
# ODE integration
  out=ode(y=yini,times=times,func=bioreactor_3,parms=NULL)
```

`bioreactor_3` is similar to `bioreactor_2` in Listing 1.3, except that the fluxes and the derivatives are programmed as vectors (Listing 1.4).

```
  bioreactor_3=function(t,y,parms) {
#
# Assign state variables:
  xylose      =y[1];
  xylitol     =y[2];
  xylulose    =y[3];
  acetaldehyde=y[4];
  ethanol     =y[5];
  acetate     =y[6];
  glycerol    =y[7];
#
# Compute fluxes
  J=rep(0,n)
  J[1]=k1*xylose;
  J[2]=k2*xylitol-km2*xylulose*ethanol;
  J[3]=k3*xylulose-km3*acetaldehyde*ethanol;
  J[4]=k4*acetaldehyde;
  J[5]=k5*acetaldehyde;
  J[6]=k6*xylulose;
#
# Time derivatives
  f=rep(0,n)
  f[1]=-J[1];
  f[2]=J[1]-J[2];
  f[3]=J[2]-2*J[3]-2*J[6];
  f[4]=3*J[3]-J[4]-J[5];
  f[5]=J[4];
  f[6]=J[5];
  f[7]=3*J[6];
#
# Calls to bioreactor_3
  ncall <<- ncall+1
#
# Return derivative vector
  return(list(c(f)))
}
```

Listing 1.4: ODE routine with vectors to facilitate the ODE programming.

We can note the following details about `bioreactor_3`.

- The sizes of the vectors for the fluxes and derivatives are declared using the R utility `rep` before the vectors are used.

```
#
# Compute fluxes
  J=rep(0,n)
       .
       .
       .
#
# Time derivatives
  f=rep(0,n)
```

  This is in contrast with some other programming languages that dynamically allocate memory as arrays are defined (first used). Thus, we might say that the `rep` "preallocates" the vectors `J` and `f`. In this case, initial values of the `n` elements of `J` and `f` are set to zero, then reset with the subsequent programming.
- The derivative vector `f` is returned from `bioreactor_3` (to `lsoda` in `ode`) as a list (as required by the ODE integrators in `deSolve`).

```
#
# Return derivative vector
  return(list(c(f)))
}
```

The intent of this example is to demonstrate how vectors can be used in programming ODEs, which can be particularly useful as the number of ODEs increases. As expected, the numerical and graphical outputs are the same as in the preceding discussion.

1.7 ODE Integrator Selection

As mentioned previously, the R utility `ode` is based (as a default) on the `lsoda` ODE integrator which has the distinguishing feature

of switching between stiff and nonstiff algorithms.[1] Another variation that is quite important for stiff ODE systems is the use of sparse matrix algorithms to conserve storage (memory) and enhance computational efficiency.

Here, we will not consider sparse matrix methods other than to point out that they have been implemented in an ODE integrator `lsodes` where the final "s" in this name designates sparse. `lsodes` is part of the `deSolve` library of ODE integrators and can be readily accessed. To illustrate how this is done, the previous main program in Listing 1.2 can be modified with the statements for the ODE integration.

```
#
# ODE integration
# out=   ode(y=yini,times=times,func=bioreactor_4,
   parms=NULL,method="lsodes")
  out=lsodes(y=yini,times=times,func=bioreactor_4,
     parms=NULL)
```

Listing 1.5: Modification of Listing 1.2 to call ODE integrator `lsodes`.

We can note the following details about this code.

- In the call to `ode` (inactive or commented), `lsodes` is called as `method="lsodes"`. This form of argument can be applied to an extensive set of ODE integrators in `ode`, both stiff and nonstiff.
- In the second line (without `ode`), `lsodes` is called explicitly.

As expected, both forms give the same numerical output as in Table 1.4. Also, they are based on the default options for `lsodes`. A variety of options are available, particularly pertaining to sparse matrix ODE integration that might be very effective as the number of ODEs increases. The details of these options are given in the R

[1]The concepts of stiffness and explicit integration are discussed in detail in [6]; Appendix C, [8].

documentation for `lsodes` and in [10]. The ODE routine called by `lsodes`, `bioreactor_4`, is the same as `bioreactor_3` in Listing 1.4 (the number was changed just to keep each case of the programming distinct and self-contained).

1.8 Euler Method

We have so far numerically integrated eqs. (1.1) by using a library integrator in `deSolve`. The integrators that are available in this way are of high quality and well established. However, we do not have access to the source code of these integrators and therefore the programmed details of the numerical integration are not available explicitly. We, therefore, now consider the programing of some classic ODE integration algorithms mainly to demonstrate how systems of ODEs can be integrated numerically. In other words, the discussion of numerical ODE integration to follow is intended to be introductory and instructional, and thereby give some insight into the computation performed by library integrators such as those in `deSolve`.[2]

The Taylor series is the starting point for most numerical integrators. We illustrate this approach starting with the most basic of all ODE integrators, the Euler method. A single ODE with an IC is considered in the following development.

$$\frac{dy}{dt} = f(y,t);\ y(t = t_0) = y_0 \qquad \text{(1.3a),(1.3b)}$$

where the derivative function $f(y,t)$ and the IC y_0 at t_0 are specified for a particular ODE problem.

The solution of eq. (1.3a) is expressed as a Taylor series at point i

$$y_{i+1} = y_i + \frac{dy_i}{dt}h + \frac{d^2y_i}{dt^2}\frac{h^2}{2!} + \cdots$$

where $h = t_{i+1} - t_i$. We can truncate this series after the linear term in h

$$y_{i+1} \approx y_i + \frac{dy_i}{dt}h \qquad (1.4)$$

[2]Numerical methods for initial-value ODEs are discussed in [2–5 and 9].

and use this approximation to step along the solution from y_0 to y_1 (with $i = 0$), then from y_1 to y_2 (with $i = 1$), etc., for a specified integration step h. This is the famous Euler method, the most basic of all ODE numerical integration methods. Note that eq. (1.4) requires only dy_i/dt, which is available from eq. (1.3a). The starting value for this stepping procedure, y_0, is available from eq. (1.3b).

Eq. (1.4) is implemented in the following variation of Listing 1.2 (Listing 1.6a) in which the ODE integration via `ode` is replaced with in-line coding of eq. (1.4).

```
#
# ODE routine
  setwd("c:/R/bme_ode/chap1")
  source("bioreactor_5.R")
#
# Parameter values for BP10001
  k1=8.87e-03;
  k2=13.18;
  k3=0.129;
  k4=0.497;
  k5=0.027;
  k6=0.545e-3;
  km2=87.7;
  km3=99.9;
#
# Initial condition
  n=7;nout=51;t=0;ncall=0;
  y=c(0.10724,0,0,0,0,0,0)
  cat(sprintf(
  "\n        t      y1      y2      y3      y4      y5
     y6      y7"))
  cat(sprintf("\n %8.0f%8.4f%8.4f%8.4f%8.4f%8.4f%8.4f
     %8.4f",
  t,y[1],y[2],y[3],y[4],y[5],y[6],y[7]))
#
# Arrays for output
  out=matrix(0,nrow=nout,ncol=(n+1))
  out[1,-1]=y
  out[1,1]=t
#
```

```
# Parameters for t integration
# nt=400;h=0.10  # unstable
  nt=800;h=0.05  # stable
#
# Euler integration
  for(i1 in 2:nout){
#
#   nt Euler steps
    for(i2 in 1:nt){
      yt=bioreactor_5(t,y);
      y=y+yt*h; t=t+h;
    }
#
#   Solution after nt Euler steps
    cat(sprintf("\n %8.0f%8.4f%8.4f%8.4f%8.4f%8.4f%8.4f
       %8.4f",
    t,y[1],y[2],y[3],y[4],y[5],y[6],y[7]))
    out[i1,-1]=y
    out[i1,1]=t
  }
#
# Calls to bioreactor_5
  cat(sprintf("\n ncall = %5d\n\n",ncall))
#
# Single plot
  par(mfrow=c(1,1))
#
# y1
  plot(out[,1],out[,2],type="l",xlab="t",ylab="y1(t),...,
     y7(t)",
    xlim=c(0,2000),ylim=c(0,0.14),lty=1, main="y1(t),...,
       y7(t) vs t",
    lwd=2)
#
# y2
  lines(out[,1],out[,3],type="l",lty=2,lwd=2)
#
# y3
  lines(out[,1],out[,4],type="l",lty=3,lwd=2)
#
# y4
```

```
  lines(out[,1],out[,5],type="l",lty=4,lwd=2)
#
# y5
  lines(out[,1],out[,6],type="l",lty=5,lwd=2)
#
# y6
  lines(out[,1],out[,7],type="l",lty=6,lwd=2)
#
# y7
  lines(out[,1],out[,8],type="l",lty=7,lwd=2)
```

Listing 1.6a: Main program with in-line explicit Euler method.

We can note the following details about Listing 1.6a.

- `reactor_5.R` is the ODE routine (rather than `reactor_3.R`, `reactor_4.R`). The differences in `reactor_5.R` are considered subsequently.

```
#
# ODE routine
  setwd("c:/R/bme_ode/chap1")
  source("bioreactor_5.R")
```

- The section for setting the parameters `k1` to `km3` is the same as in Listings 1.1 and 1.2 and is therefore not repeated here.
- The IC is placed in vector `c`, then displayed with a heading.

```
#
# Initial condition
  n=7;nout=51;t=0;ncall=0;
  y=c(0.10724,0,0,0,0,0,0)
  cat(sprintf(
  "\n        t      y1      y2      y3      y4      y5
     y6      y7"))
  cat(sprintf("\n %8.0f%8.4f%8.4f%8.4f%8.4f%8.4f%8.4f
     %8.4f",
  t,y[1],y[2],y[3],y[4],y[5],y[6],y[7]))
```

Note in particular the number of ODEs, `n=7`, and the number of output points `nout=51`.

- An array `out` is defined for the numerical solution that has the same format as `out` in Listings 1.1 and 1.2.

```
#
# Arrays for output
  out=matrix(0,nrow=nout,ncol=(n+1))
  out[1,-1]=y
  out[1,1]=t
```

To further explain this programming,

— `out` is defined as a 2D array with the R utility `matrix`. `outs` has `nout` (= 51) rows for the output points in t set previously and `(n+1)` $= 7 + 1$ columns for the seven dependent variables of eqs. (1.1) and the independent variable t.

```
out=matrix(0,nrow=nout,ncol=(n+1))
```

— The first index of `out` is `1` corresponding to $t = 0$.

— The second index (subscript) of `out` indicates all values except 1, that is, -1. In this case, there are seven values corresponding to the ICs for `y` set previously.

```
out[1,-1]=y
```

— The initial value of $t(= 0)$ is placed in `out` with the second subscript set to `1`.

```
out[1,1]=t
```

- The integration step h in eq. (1.4) is set numerically. This is typically done by some trial and error. For example, because $0 \leq t \leq 2000$, $h = 0.1$ corresponds to $(2000)/(0.1) = 20,000$ Euler steps. However, when this value is used, the solution is unstable as will be demonstrated subsequently. With $h = 0.05$, the solution is stable (and as we will observe, also accurate). In other words, h generally has to be selected so that the numerical solution is stable and accurate.

```
#
# Parameters for t integration
```

```
# nt=400;h=0.10  # unstable
  nt=800;h=0.05  # stable
```

Once h is defined, the number of Euler steps, `nt`, to span the output interval of 40 is `nt` $= 40/0.05 = 800$. In other words, 800 Euler steps according to eq. (1.4) are completed for each output at $t = 40, 80, \ldots, 2000$. This is in contrast with the variable step integrators in `ode` which adjust h to achieve a prescribed accuracy of the numerical solution and therefore may take many fewer steps than 800 for each output interval 40. For example, `lsoda` required `ncall = 427` (Table 1.4) calls to the ODE integrator, whereas the Euler integrator of Listing 1.6a requires 2000/0.05 = 40,000 calls to the ODE routine, `bioreactor_5`. This larger number (40,000 rather than 427) is a manifestation of stiffness of eqs. (1.1). In other words, the Euler method requires 40,000 steps to main stability of the numerical solution rather than accuracy. Thus, the effectiveness of the stiff integrator of `lsoda` for this example (eqs. (1.1)) is clear.

- The Euler integration proceeds with two `for` loops.

```
#
# Euler integration
  for(i1 in 2:nout){
#
#   nt Euler steps
    for(i2 in 1:nt){
      yt=bioreactor_5(t,y);
      y=y+yt*h; t=t+h;
    }
```

The first loop with index `i1` steps through the 50 output points $t = 40, 80, \ldots, 2000$ (after $t = 0$). The second loop with index `i2` steps through the 800 Euler steps for each output (so that there are a total of $50 \times 800 = 40,000$ Euler steps for the complete solution to $t = 2000$). Within this second loop, the derivative vector dy_i/dt in eq. (1.4) is computed by a call to the ODE routine `bioreactor_5` (`yt` has seven elements but the vector facility of R is used so that subscripting is not required). Then,

the solution is advanced from i to $i + 1$ according to eq. (1.4). Also, t is advanced by the integration step h, that is, $t_{i+1} = t_i + h$.

- After each pass of the loop in `i1`, the solution is put into array `out` for subsequent plotting.

```
#
#   Solution after nt Euler steps
    cat(sprintf("\n %8.0f%8.4f%8.4f%8.4f%8.4f%8.4f
       %8.4f%8.4f",
    t,y[1],y[2],y[3],y[4],y[5],y[6],y[7]))
    out[i1,-1]=y
    out[i1,1]=t
  }
```

Note the use of index `i1` when writing `out`. The final `}` completes the loop in `i1`.

- Since array `out` is used in the same way as in Listings 1.1, 1.2, and 1.5 (with `ode` and `lsodes`), the plotting used previously can be used again (listed above but not here).

`bioreactor_5` called by the main program in Listing 1.6a is in Listing 1.6b. It is the same as `bioreactor_3` and `bioreactor_4` except for the final line that returns the derivative vector `f`.

```
  bioreactor_5=function(t,y,parms) {
    .                              .
    .                              .
  (same as bioreactor_3, bioreactor_4)
    .                              .
    .                              .
#
# Return derivative vector
  return(c(f))
}
```

Listing 1.6b: `bioreactor_5.R` called in Listing 1.6a.

The difference in the return statements

```
From Listing 1.4
```

```
return(list(c(f))

From Listing 1.6b
return(c(f))
```

is small but important.

For Listing 1.4, `return(list(c(f))` returns the derivative vector `f` as a list that is required by the integrators in `deSolve`, for example, `ode`, `lsodes`. For Listing 1.6b, `return(c(f))` returns the derivative vector `f` as a numerical vector, which is required in the programming of the Euler method in Listing 1.6a. In particular, the arithmetic multiplication `*` used in `y=y+yt*h` must operate on two numerical objects, in this case `yt` and `h`. If `yt` is a list rather than a numerical vector, the multiplication `*` will not function (an error message results). While this may seem like a minor detail, the distinction between a list and a numerical object must be taken into consideration.

Abbreviated numerical output from Listings 1.6a and 1.6b is in Table 1.5.

This output is identical to the output in Table 1.4 to four figures, for example, at $t = 2000$,

```
Table 1.4

  2000  0.0000  0.0002  0.0003  0.0000  0.1307  0.0071
     0.0223

  ncall =   427

Table 1.5

  2000  0.0000  0.0002  0.0003  0.0000  0.1307  0.0071
     0.0223

  ncall = 40000
```

Thus, we conclude that the Euler method reproduces the output from `ode` and `lsodes` for eqs. (1.1), which can be considered a check on the numerical solutions. However, an essential difference in the programming was the need to provide an integration

TABLE 1.5 Numerical output from Listings 1.6a and 1.6b, $h = 0.05$.

```
   t       y1       y2       y3       y4       y5       y6       y7
   0   0.1072   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000
  40   0.0752   0.0020   0.0153   0.0009   0.0195   0.0011   0.0006
  80   0.0527   0.0053   0.0221   0.0008   0.0361   0.0020   0.0018
 120   0.0370   0.0081   0.0246   0.0006   0.0497   0.0027   0.0034
 160   0.0259   0.0101   0.0248   0.0005   0.0609   0.0033   0.0050
 200   0.0182   0.0112   0.0239   0.0004   0.0702   0.0038   0.0066
 240   0.0128   0.0116   0.0224   0.0004   0.0780   0.0042   0.0081
                  .                             .
                  .                             .
                  .                             .
                 Output for t = 280 to 1720 removed
                  .                             .
                  .                             .
                  .                             .
1760   0.0000   0.0004   0.0005   0.0000   0.1303   0.0071   0.0222
1800   0.0000   0.0004   0.0004   0.0000   0.1303   0.0071   0.0222
1840   0.0000   0.0003   0.0004   0.0000   0.1304   0.0071   0.0222
1880   0.0000   0.0003   0.0004   0.0000   0.1305   0.0071   0.0223
1920   0.0000   0.0003   0.0003   0.0000   0.1306   0.0071   0.0223
1960   0.0000   0.0003   0.0003   0.0000   0.1306   0.0071   0.0223
2000   0.0000   0.0002   0.0003   0.0000   0.1307   0.0071   0.0223

ncall = 40000
```

step (h in eq. (1.4)) for the Euler method, whereas `ode` and `lsodes` automatically adjusted the step in accordance with the default error tolerances for `ode` (`lsoda`) and `lsodes`. We next consider some additional consequences of using eq. (1.4).

1.9 Accuracy and Stability Constraints

We return to the important detail in Listing 1.6a pertaining to the value of the integration step h and the stability of the numerical solution.

```
#
# Parameters for t integration
# nt=400;h=0.10  # unstable
  nt=800;h=0.05  # stable
```

This limit on the value of h reflects the stability limit of the explicit Euler method in eq. (1.4) where explicit refers to the direct (explicit) calculation of y_{i+1} from y_i. This calculation is straightforward but also has a limit on h for the calculation to remain stable (as demonstrated by the transition from $h = 0.05$ to $h = 0.10$).

To further investigate this limit with some basic ideas from linear algebra [7, Appendix 2], we consider a $n \times n$ linear, constant coefficient ODE system for which there will be an associated set of n eigenvalues. The ODE system will be stable (have a stable solution) if and only if (iff) all of the eigenvalues are in the left half of the complex plane, that is, iff the real parts of the eigenvalues are nonpositive. However, even with a stable ODE system, the numerical solution from an explicit algorithm such as the Euler method of eq. (1.4) can be unstable unless the integration step h is restricted. For the explicit Euler method, this stability limit is [4, p 230]

$$|\lambda h| \leq c \tag{1.5}$$

with $c = 2$ for each eigenvalue λ; if the eigenvalue is complex, then $|\ |$ denotes a modulus (absolute value).

Eq. (1.5) indicates that the maximum step h for a stable solution is set by the eigenvalue with the largest modulus, $\lambda_{\max}$. However, the timescale for the ODE system, for example, $0 \leq t \leq 2000$ of eqs. (1.1), is determined by the eigenvalue with the smallest modulus, $\lambda_{\min}$. Thus, the ratio

$$\text{SR} = \frac{\lambda_{\max}}{\lambda_{\min}} \tag{1.6}$$

termed the *stiffness ratio* (SR), is an indicator of the number of steps required to compute a complete solution to an ODE system with an explicit integrator such as eq. (1.4). As a qualitative guideline, the following Table 1.6 provides approximate ranges of values of SR.

TABLE 1.6 Qualitative degree of stiffness.

SR	Stiffness
<100	nonstiff
$100 \leq SR \leq 1000$	moderately stiff
>1000	stiff

The effect of an increase in SR in Table 1.6 can be interpreted as increased stiffness with the spread (spectrum, spectral radius) of the ODE eigenvalues, that is, for a stiff ODE system, $|\lambda_{\max}| >> |\lambda_{\min}|$. The requirement of using $h = 0.05$ (and not $h = 0.1$) with $(2000)(20) = 40,000$ Euler steps for a complete solution (in Listing 1.6a) suggests eqs. (1.1) are effectively stiff.

However, there is one additional complication. The preceding discussion of the SR based on eigenvalues presupposes a linear constant coefficient ODE system. But eqs. (1.1) are nonlinear, so the use of the concept of eigenvalues is not straightforward. We will just conclude that if an ODE systems requires a large number of integration steps for a complete numerical solution, the ODE system is effectively stiff and therefore requires a stiff integrator to produce a solution with a modest number of steps.

Then we have to consider what is a stiff integrator. The general answer is that it is implicit rather than explicit. For example, rather than use the explicit Euler method of eq. (1.4), we can use the implicit form

$$y_{i+1} \approx y_i + \frac{dy_{i+1}}{dt} h \tag{1.7}$$

The only difference between eqs. (1.4) and (1.7) is the point along the solution at which the derivative dy/dt is evaluated (i for eq. (1.4) and $i + 1$ for eq. (1.7)). While this may seem like a minor point, it is important for at least two reasons.

- The constant c in eq. (1.5) is ∞. In other words, the implicit Euler method is unconditionally stable and, therefore, there is no stability restriction placed on h (the only restriction is from accuracy).

- Since y_{i+1} appears on both the sides of eq. (1.7), it is implicit in the solution at the next point along the solution. For systems of ODEs, this means that the solution of a system of simultaneous algebraic equations is required to move from point i to point $i + 1$ along the solution. If the ODEs are nonlinear, a system of nonlinear algebraic equations must be solved numerically which can be a formidable requirement (depending on the number of ODEs and the form of their nonlinearities). Generally a variant of Newton's method is used to solve the nonlinear algebraic system.

This discussion indicates that the stability limit of explicit methods can be circumvented by using an implicit method, but this increased stability comes at the cost of substantially increased computational complexity.

Two other points should be mentioned.

- Because implicit (stiff) integrators require rather substantial computation, they should be used only if the ODE system is actually stiff. In other words, a nonstiff (explicit) integrator should be tried first, and if it requires a small step (as in Listing 1.6a) to maintain stability, a stiff (implicit) integrator should be considered.
- Nonlinearity in an ODE system does not necessarily mean that the ODEs are stiff (a common misconception is to equate nonlinearity and stiffness).

These two points indicate that the choice of a stiff versus a nonstiff ODE integrator may not be straightforward and clear cut. To assist with this requirement, `lsoda`, the default integrator of `ode` switches automatically between stiff and nonstiff options as the solution proceeds; the "a" denotes the automatic switching, which is based on the eigenvalue analysis. The details of `lsoda` are rather complicated, but the final result is a well-established quality integrator that can be used, for example, in `ode` without becoming involved in the details.

To conclude this discussion of the Euler method, if the integration is done with $h = 0.10$ rather than $h = 0.05$, the output from Listing 1.6a (abbreviated) is given in Table 1.7.

TABLE 1.7 Numerical output from Listings 1.6a and 1.6b, $h = 0.10$.

```
   t       y1       y2       y3       y4       y5       y6       y7
   0   0.1072   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000
  40   0.0752   0.0020   0.0153   0.0009   0.0195   0.0011   0.0006
  80   0.0527   0.0053   0.0221   0.0008   0.0361   0.0020   0.0018
 120   0.0370   0.0081   0.0246   0.0006   0.0497   0.0027   0.0034
 160   0.0259   0.0101   0.0248   0.0005   0.0609   0.0033   0.0050
 200   0.018294287.4828-211142.0815175269.8277 11.5836 0.6293
                                                          -0.0091
 240   0.0127      NaN      NaN      NaN      NaN      NaN      NaN
 280   0.0089      NaN      NaN      NaN      NaN      NaN      NaN
 320   0.0063      NaN      NaN      NaN      NaN      NaN      NaN
 360   0.0044      NaN      NaN      NaN      NaN      NaN      NaN
 400   0.0031      NaN      NaN      NaN      NaN      NaN      NaN
       .                                              .
       .                                              .
       .                                              .
           Output from t = 440 to 1760 deleted
       .                                              .
       .                                              .
       .                                              .
1800   0.0000      NaN      NaN      NaN      NaN      NaN      NaN
1840   0.0000      NaN      NaN      NaN      NaN      NaN      NaN
1880   0.0000      NaN      NaN      NaN      NaN      NaN      NaN
1920   0.0000      NaN      NaN      NaN      NaN      NaN      NaN
1960   0.0000      NaN      NaN      NaN      NaN      NaN      NaN
2000   0.0000      NaN      NaN      NaN      NaN      NaN      NaN

ncall = 20000
```

We can note the following details about this output.

- The solutions starts at the same ICs ($t = 0$) as in Table 1.5.

- At $t = 200$, the numbers (not the correct numerical solution) suddenly change, reflecting an instability in the calculations.
- For the remaining values of t, `NaN` (not a number) indicates the calculations have failed.
- The number of calls to `bioreactor_5` is 20,000, as expected (because the integration h is doubled from 0.05 to 0.01). But clearly, the doubling has violated the stability criterion (1.5).

In summary, `lsoda` performed 427 derivative evaluations while the explicit Euler integrator performed $(2000)(20) = 40,000$, a difference of nearly two orders of magnitude (a factor of 10^2). This example clearly indicates the effectiveness of a stiff integrator (in `lsoda`). However, we again indicate that a nonstiff integrator should be tried first in case the ODE system is not stiff (and therefore the increased computations for each step of a stiff integrator are not required).

The integration step for explicit integrators may be constrained by accuracy and/or stability. In the case of eqs. (1.1), h is constrained by stability as reflected in Listing 1.6a and apparently not by accuracy (recall the agreement of the solutions in Tables 1.4 and 1.5). However, the explicit Euler method of eq. (1.4) can be constrained by accuracy because it is only first-order correct ($O(h)$). Therefore, the explicit integrators we discuss next are worth considering for a new nonstiff ODE problem application because they are of higher order than the first-order explicit Euler method of eq. (1.4). However, higher order explicit methods do not have improved stability. For example, c in eq. (1.5) generally does not exceed 3 even for higher order explicit methods.

As a point of notation, $O(h)$ used above denotes "of first order in h" or "first-order correct" and is intended to indicate the error resulting from the truncated Taylor series. For example, the Taylor series is truncated after the h term to give the Euler method. This can be stated alternatively, the truncation error is

$$\text{error} = O(h) = c_t h$$

where c_t is taken as a constant (generally it will vary along the solution). To demonstrate that the Euler method is first order, the reader might compute the numerical solution of an ODE with an exact solution (eqs. (1.1) does not have a known exact solution), then use the exact error (as the absolute value of the difference between the Euler solution and the exact solution at a particular t) for a series of h values in a plot of the exact error versus h. This plot, for sufficiently small h, will be a straight line with slope 1. For an nth order method with $O(h^n)$, a plot in log-log format will be a line with slope n since

$$\text{error} = O(h^n) = c_t h^n$$

or

$$\log(\text{error}) = n\log(h) + \log(c_t)$$

This type of graphical error analysis is usually done with the model problem (a special case of eq. (1.3))

$$\frac{dy}{dt} = \lambda y;\ y(0) = y_0$$

and the exact solution

$$y(t) = y_0 e^{\lambda t}$$

Note that for the eigenvalue $\lambda \leq 0$, the exact solution is stable. Therefore, this test problem can also be used to test the Euler stability criterion of eq. (1.5) (with $c = 2$) by using a value of h greater than $2/|\lambda|$. The calculations for this accuracy and stability analysis can be carried out with a straightforward modification of Listings 1.6a and 1.6b (routines for this purpose are provided with the software download for this book).

If the ODE system is stiff and therefore the integration step of an explicit method is constrained by stability, this constraint can usually be circumvented by using an implicit method such as the implicit Euler method of eq. (1.7); but the implicit Euler method of eq. (1.7) is only first order so that a higher order implicit method is generally used

to give both good accuracy and stability, for example, the stiff integrator in `lsoda`. Implicit methods have the cost of additional calculations for the solution of systems of nonlinear algebraic equations, but the additional calculations are worthwhile for stiff systems (recall again the improvement of two orders of magnitude between the explicit Euler method of eq. (1.4) and the higher order implicit method of `lsoda`).

1.10 Modified Euler Method as a Runge–Kutta Method

The preceding discussion of the explicit Euler method of eq. (1.4) indicated the limited first-order accuracy (the error is $O(h)$) resulting from the truncation of the Taylor series after the h term. We now consider how this order can be increased, basically by using additional terms in the Taylor series.

$$y_{i+1} = y_i + \frac{dy_i}{dt}h + \frac{d^2y_i}{dt^2}\frac{h^2}{2!} + \cdots \tag{1.8}$$

For example, we might consider including the h^2 term. To do this, we have to use the second derivative d^2y_i/dt^2 which is generally unavailable because the ODE, eq. (1.3a), provides only the first derivative. One possibility would be to repeatedly differentiate the derivative function (RHS of eq. (1.3a)) to obtain the higher order derivatives that are required as more terms are included in the Taylor series. But this quickly becomes impractical if we are considering an ODE system. In other words, we need to have a procedure for including the higher order terms in the Taylor series without having to differentiate the ODEs. We now consider how this can be done with the Runge–Kutta method.

If the second derivative is approximated as the finite difference of the first derivative,

$$\frac{d^2y_i}{dt^2} \approx \frac{dy_{i+1}/dt - dy_i/dt}{h} \tag{1.9}$$

then substitution of eq. (1.9) in eq. (1.8) with truncation after the h^2 term gives

$$y_{i+1} = y_i + \frac{dy_i}{dt}h + \frac{\frac{dy_{i+1}}{dt} - \frac{dy_i}{dt}}{h}\frac{h^2}{2!}$$
$$= y_i + \frac{\frac{dy_{i+1}}{dt} + \frac{dy_i}{dt}}{2}h \tag{1.10}$$

Eq. (1.10) indicates that the derivatives at the base point i and the advanced point $i+1$ are averaged in stepping along the solution (from y_i to y_{i+1}). The derivative y_{i+1}/dt can be computed by substituting y_{i+1} from eq. (1.4) in the ODE, eq. (1.3a). This leads to an algorithm with the following steps:

- Starting at the base point y_i, t_i, the ODE, eq. (1.3a), is used to calculate dy_i/dt
$$\frac{dy_i}{dt} = f(y_i, t_i) \tag{1.11a}$$
- Eq. (1.4) is used to calculate y_{i+1} (with $t_{i+1} = t_i + h$) from the result of eq. (1.11a).
$$y_{i+1} = y_i + \frac{dy_i}{dt}h \tag{1.11b}$$
- The ODE, eq. (1.3a), is used to calculate dy_{i+1}/dt from the result of eq. (1.11b).
$$\frac{dy_{i+1}}{dt} = f(y_{i+1}, t_i + h) \tag{1.11c}$$
- Eq. (1.10) is used to calculate an improved y_{i+1} from the results of eqs. (11.1a) and (1.11c).
$$y_{i+1} = y_i + \frac{\frac{dy_{i+1}}{dt} + \frac{dy_i}{dt}}{2}h \tag{1.11d}$$

Eqs. (1.11) are the modified Euler method (also termed the *extended Euler method* or *Heun's method*). Also, if we consider y_{i+1} from eq. (1.11b) to be a predicted value, and y_{i+1} from eq. (1.11d) to be a corrected value, eqs. (1.11) can be considered as

a predictor–corrector scheme. An important point to note is that eqs. (1.11) do not require differentiating the ODE, eq. (1.3a) (only the ODE is used), but the final result of eq. (1.11d) is second-order correct (the error is $O(h^2)$). The cost of this improved accuracy ($O(h)$ of the Euler method, eq. (1.4), improved to $O(h^2)$ of the modified Euler method, eqs. (1.11)), is an increase of one derivative evaluation in eq. (1.4) to two derivatives evaluation in eqs. (1.11). Usually, the increased accuracy is well worth the additional calculational effort (additional derivative evaluations from the ODE).

In other words, we have achieved higher order accuracy without differentiating the ODE by evaluating the derivative from the ODE at selected points along the solution. In the case of the modified Euler method, the selected points for derivative evaluation are (y_i, t_i) and (y_{i+1}, t_{i+1}) as reflected in eq. (1.11d). This idea of multiple evaluation of the ODE derivative function along selected points of the numerical solution to produce higher order methods is the basis of the Runge–Kutta method.[3] Therefore, we restate eqs. (1.11) in the Runge–Kutta format so that we can then logically extend them to higher order methods stated in the Runge–Kutta format.

$$k_1 = f(y_i, t_i)h \tag{1.12a}$$

$$y_{i+1} = y_i + k_1 \tag{1.12b}$$

$$k_2 = f(y_{i+1}, t_{i+1})h = f(y_i + k_1, t_i + h)h \tag{1.12c}$$

$$y_{i+1} = y_i + \frac{k_1 + k_2}{2}h \tag{1.12d}$$

Note that the derivatives (multiplied by h) are given the names k_1, k_2 by convention.

We now consider the programming of eqs. (1.11) and (1.12) (two cases). A main program with these equations is in Listing 1.7.

```
#
# ODE routine
  setwd("c:/R/bme_ode/chap1")
```

[3]The Runge–Kutta methods are discussed in [2] and [3].

```
  source("bioreactor_6.R")
#
# Select modified Euler format
#
# ncase = 1: Taylor series format
#
# ncase = 2: Runge Kutta format
  ncase=1;
#
# Parameter values for BP10001
  k1=8.87e-03;
  k2=13.18;
  k3=0.129;
  k4=0.497;
  k5=0.027;
  k6=0.545e-3;
  km2=87.7;
  km3=99.9;
#
# Initial condition
  n=7;nout=51;t=0;ncall=0;
  y=c(0.10724,0,0,0,0,0,0)
  cat(sprintf(
  "\n        t      y1      y2      y3      y4      y5
     y6      y7"))
  cat(sprintf("\n %8.0f%8.4f%8.4f%8.4f%8.4f%8.4f%8.4f
     %8.4f",
  t,y[1],y[2],y[3],y[4],y[5],y[6],y[7]))
#
# Arrays for output
  out=matrix(0,nrow=nout,ncol=(n+1))
  out[1,-1]=y
  out[1,1]=t
#
# Parameters for t integration
# nt=400;h=0.10  # unstable
  nt=800;h=0.05  # stable
#
# Modified Euler integration
  for(i1 in 2:nout){
#
```

```
#   nt modified Euler steps
    for(i2 in 1:nt){
    if(ncase==1){
      yb=y
      ytb=bioreactor_6(t,y)
      y=yb+ytb*h; t=t+h
      yt=bioreactor_6(t,y)
      y=yb+(ytb+yt)/2*h
    }
    if(ncase==2){
      yb=y
      rk1=bioreactor_6(t,y)*h
      y=yb+rk1; t=t+h
      rk2=bioreactor_6(t,y)*h
      y=yb+(rk1+rk2)/2
    }
    }
#
#   Solution after nt modified Euler steps
    cat(sprintf("\n %8.0f%8.4f%8.4f%8.4f%8.4f%8.4f%8.4f
       %8.4f",
    t,y[1],y[2],y[3],y[4],y[5],y[6],y[7]))
    out[i1,-1]=y
    out[i1,1]=t
  }
#
# Calls to bioreactor_6
  cat(sprintf("\n ncall = %5d\n\n",ncall))
#
# Single plot
  par(mfrow=c(1,1))
#
# y1
  plot(out[,1],out[,2],type="l",xlab="t",ylab="y1(t),...,
     y7(t)",
    xlim=c(0,2000),ylim=c(0,0.14),lty=1, main="y1(t),...,
       y7(t) vs t",
    lwd=2)
#
# y2
  lines(out[,1],out[,3],type="l",lty=2,lwd=2)
```

```
#
# y3
  lines(out[,1],out[,4],type="l",lty=3,lwd=2)
#
# y4
  lines(out[,1],out[,5],type="l",lty=4,lwd=2)
#
# y5
  lines(out[,1],out[,6],type="l",lty=5,lwd=2)
#
# y6
  lines(out[,1],out[,7],type="l",lty=6,lwd=2)
#
# y7
  lines(out[,1],out[,8],type="l",lty=7,lwd=2)
```

Listing 1.7: Main program with the in-line modified Euler method.

Listing 1.7 is similar to Listing 1.6a for the Euler method but it is included here because of some of the following significant differences.

- The ODE routine is `bioreactor_6`, which is the same as `bioreactor_5` of Listing 1.6b

  ```
  #
  # ODE routine
    setwd("c:/R/bme_ode/chap1")
    source("bioreactor_6.R")
  #
  # Select modified Euler format
  #
  # ncase = 1: Taylor series format
  #
  # ncase = 2: Runge Kutta format
    ncase=1;
  ```

  `ncase` has the values 1 for eqs. (1.11) or 2 for eqs. (1.12).
- The parameters for `BP1001` of Listing 1.6a are used again.

- The ICs of Listing 1.6a are used again. Note in particular that the counter `ncall` is initialized.

```
#
# Initial condition
  n=7;nout=51;t=0;ncall=0;
  y=c(0.10724,0,0,0,0,0,0)
  cat(sprintf(
  "\n        t      y1      y2      y3      y4      y5
     y6      y7"))
  cat(sprintf("\n %8.0f%8.4f%8.4f%8.4f%8.4f%8.4f%8.4f
     %8.4f",
  t,y[1],y[2],y[3],y[4],y[5],y[6],y[7]))
```

- The array `out` for the solution again has the same format as produced by the integrators in `deSolve`.

```
#
# Arrays for output
  out=matrix(0,nrow=nout,ncol=(n+1))
  out[1,-1]=y
  out[1,1]=t
#
# Parameters for t integration
# nt=400;h=0.10  # unstable
  nt=800;h=0.05  # stable
```

  The integration step is $h = 0.05$, which is stable (the value of c in eq. (1.5) is again 2 for the modified Euler method [4, p 230, Fig. 6.2]).

- The programming of eqs. (1.11) (`ncase = 1`) is similar to the programming of the Euler method of Listing 1.6a.

```
#
# Modified Euler integration
  for(i1 in 2:nout){
#
#   nt modified Euler steps
    for(i2 in 1:nt){
    if(ncase==1){
```

```
   yb=y
   ytb=bioreactor_6(t,y)
   y=yb+ytb*h; t=t+h
   yt=bioreactor_6(t,y)
   y=yb+(ytb+yt)/2*h
 }
```

The correspondence of the coding with eqs. (1.11) (for `ncase=1`) is clear. Note that two calls to the ODE routine, `bioreactor_6`, are used to achieve the second-order accuracy of the modified Euler method as explained previously.

- The programming of eqs. (1.12) (`ncase=2`) is straightforward. k_1 and k_2 in eqs. (1.12) are programmed as `rk1` and `rk2` to avoid a conflict with the kinetic rate constants `k1,k2` defined previously.

```
if(ncase==2){
   yb=y
   rk1=bioreactor_6(t,y)*h
   y=yb+rk1; t=t+h
   rk2=bioreactor_6(t,y)*h
   y=yb+(rk1+rk2)/2
 }
 }
```

- The numerical and graphical displays of the solutions are the same as in Listing 1.6a. Note also the display of the counter `ncall`.

```
#
# Calls to bioreactor_6
  cat(sprintf("\n ncall = %5d\n\n",ncall))
```

ODE routine `bioreactor_6` is not listed here because it is the same as in Listing 1.6b.

The output from these routines is the same as in Tables 1.4 and 1.5, except the number of calls to `bioreactor_6` is `80000` as expected (twice the calls for the Euler method because of two derivative evaluations in each integration step according to eqs. (1.12)).

The comparison of the output from the Euler method and the modified Euler method suggests a way to evaluate the accuracy of the numerical solution, that is, compare the numerical solutions computed by two different methods. This idea is used in the following reprogramming of the numerical integration in Listing 1.7.

```
#
# ODE routine
  setwd("c:/R/bme_ode/chap1")
  source("bioreactor_7.R")
#
# Select modified Euler format
#
# ncase = 1: RK format
#
# ncase = 2: RK format with explicit error estimate
  ncase=2;
#
# Parameter values for BP10001
  k1=8.87e-03;
  k2=13.18;
  k3=0.129;
  k4=0.497;
  k5=0.027;
  k6=0.545e-3;
  km2=87.7;
  km3=99.9;
#
# Initial condition
  n=7;nout=51;t=0;ncall=0;
  y=c(0.10724,0,0,0,0,0,0)
  cat(sprintf(
  "\n        t      y1      y2      y3      y4      y5
     y6      y7"))
  if(ncase==2){
  ee=c(0,0,0,0,0,0,0)
    cat(sprintf(
  "\n               e1      e2      e3      e4      e5
     e6      e7"))}
  cat(sprintf("\n %8.0f%8.4f%8.4f%8.4f%8.4f%8.4f%8.4f
     %8.4f",
```

```
  t,y[1],y[2],y[3],y[4],y[5],y[6],y[7]))
  if(ncase==2){
    cat(sprintf("\n         %8.4f%8.4f%8.4f%8.4f%8.4f%8.4f
       %8.4f\n",
    ee[1],ee[2],ee[3],ee[4],ee[5],ee[6],ee[7]))}
#
# Arrays for output
  out=matrix(0,nrow=nout,ncol=(n+1))
  out[1,-1]=y
  out[1,1]=t
#
# Parameters for t integration
# nt=400;h=0.10  # unstable
  nt=800;h=0.05  # stable
#
# Modified Euler integration
  for(i1 in 2:nout){
#
#   nt modified Euler steps
    for(i2 in 1:nt){
    if(ncase==1){
      yb=y
      rk1=bioreactor_7(t,y)*h
      y=yb+rk1; t=t+h
      rk2=bioreactor_7(t,y)*h
      y=yb+(rk1+rk2)/2
    }
    if(ncase==2){
      yb=y
      rk1=bioreactor_7(t,y)*h
      y1=yb+rk1; t=t+h
      rk2=bioreactor_7(t,y1)*h
      y=yb+(rk1+rk2)/2
      ee=y-y1
    }
    }
#
#   Solution after nt modified Euler steps
    cat(sprintf("\n %8.0f%8.4f%8.4f%8.4f%8.4f%8.4f%8.4f
       %8.4f",
    t,y[1],y[2],y[3],y[4],y[5],y[6],y[7]))
```

```
    if(ncase==2){
      cat(sprintf("\n         %8.4f%8.4f%8.4f%8.4f%8.4f
         %8.4f%8.4f\n",
      ee[1],ee[2],ee[3],ee[4],ee[5],ee[6],ee[7]))}
    out[i1,-1]=y
    out[i1,1]=t
  }
#
# Calls to bioreactor_7
  cat(sprintf("\n ncall = %5d\n\n",ncall))
#
# Single plot
  par(mfrow=c(1,1))
#
# y1
  plot(out[,1],out[,2],type="l",xlab="t",ylab="y1(t),...,
     y7(t)",
    xlim=c(0,2000),ylim=c(0,0.14),lty=1, main="y1(t),...,
       y7(t) vs t",
    lwd=2)
#
# y2
  lines(out[,1],out[,3],type="l",lty=2,lwd=2)
#
# y3
  lines(out[,1],out[,4],type="l",lty=3,lwd=2)
#
# y4
  lines(out[,1],out[,5],type="l",lty=4,lwd=2)
#
# y5
  lines(out[,1],out[,6],type="l",lty=5,lwd=2)
#
# y6
  lines(out[,1],out[,7],type="l",lty=6,lwd=2)
#
# y7
  lines(out[,1],out[,8],type="l",lty=7,lwd=2)
```

Listing 1.8: Comparison of the solutions of eqs. (1.1) from the Euler and the modified Euler methods.

We can note the following details about Listing 1.8.

- The ODE routine is `bioreactor_7` and is the same as Listing 1.6b. Also, two cases are programmed. `ncase=1` is the same as the integration in Listing 1.7. `ncase=2` gives an estimate of the solution error by comparing the solutions from the Euler method (eq. (1.4)) and the modified Euler method (eqs. (1.11)).

```
#
# ODE routine
  setwd("c:/R/bme_ode/chap1")
  source("bioreactor_7.R")
#
# Select modified Euler format
#
# ncase = 1: RK format
#
# ncase = 2: RK format with explicit error estimate
  ncase=2;
```

- The programming of the parameters for eqs. (1.1) is the same as in Listing 1.7.
- The ICs of Listing 1.7 are now extended to include the estimated integration error `ee`. The initial values (at $t = 0$) of the error are zero (`ee=c(0,0,0,0,0,0,0)`) because the numerical solutions for the Euler and modified Euler methods are the same (they have the same IC, `y=c(0.10724,0,0,0,0,0,0)`).

```
#
# Initial condition
  n=7;nout=51;t=0;ncall=0;
  y=c(0.10724,0,0,0,0,0,0)
  cat(sprintf(
  "\n        t      y1      y2      y3      y4      y5
     y6      y7"))
  if(ncase==2){
  ee=c(0,0,0,0,0,0,0)
    cat(sprintf(
  "\n               e1      e2      e3      e4      e5
```

```
   e6      e7"))}
cat(sprintf("\n %8.0f%8.4f%8.4f%8.4f%8.4f%8.4f%8.4f
   %8.4f",
t,y[1],y[2],y[3],y[4],y[5],y[6],y[7]))
if(ncase==2){
  cat(sprintf("\n         %8.4f%8.4f%8.4f%8.4f%8.4f
     %8.4f%8.4f\n",
  ee[1],ee[2],ee[3],ee[4],ee[5],ee[6],ee[7]))}
```

Output with a heading and the ICs is included. Also, the initial values of the estimated error, `ee`, are displayed.

- The output array, `out`, and the parameters for the t integration are the same as in Listing 1.7.
- The two cases for the ODE integration are programmed as

```
#
# Modified Euler integration
  for(i1 in 2:nout){
#
#   nt modified Euler steps
    for(i2 in 1:nt){
    if(ncase==1){
      yb=y
      rk1=bioreactor_7(t,y)*h
      y=yb+rk1; t=t+h
      rk2=bioreactor_7(t,y)*h
      y=yb+(rk1+rk2)/2
    }
    if(ncase==2){
      yb=y
      rk1=bioreactor_7(t,y)*h
      y1=yb+rk1; t=t+h
      rk2=bioreactor_7(t,y1)*h
      y=yb+(rk1+rk2)/2
      ee=y-y1
    }
    }
```

The ODE routine is `bioreactor_7` and is the same as Listing 1.6b. For `ncase=1`, the programming is the same as in Listing 1.7.

For ncase=2, the error ee is estimated as the difference between the Euler solution, y1, and the modified Euler solution, y, that is, ee=y-y1. Note that the R vector facility is used because y1, y, ee are vectors with seven elements. The final left } concludes the for loop in i2.

- The solution and estimated errors are displayed after nt modified Euler steps (performed in the for loop with index i2).

```
#
#   Solution after nt modified Euler steps
    cat(sprintf("\n %8.0f%8.4f%8.4f%8.4f%8.4f%8.4f
       %8.4f%8.4f",
    t,y[1],y[2],y[3],y[4],y[5],y[6],y[7]))
    if(ncase==2){
      cat(sprintf("\n         %8.4f%8.4f%8.4f%8.4f
         %8.4f%8.4f%8.4f\n",
      ee[1],ee[2],ee[3],ee[4],ee[5],ee[6],ee[7]))}
    out[i1,-1]=y
    out[i1,1]=t
  }
```

The final left } concludes the for in i1.

- The number of calls to the ODE routine bioreactor_7, ncall, and the plotting are the same as in Listing 1.7.

Abbreviated output from Listing 1.8 is given in Table 1.8.

We can note the following details about this output.

- The ICs ($t = 0$) are confirmed for the solution and the estimated error.
- The estimated errors throughout the solution are zero to four figures as expected, because the Euler solution from Listing 1.6a and the modified Euler solution from Listings 1.9 and 1.10 agree to at least four figures.
- The number of calls to bioreactor_7 remains at (2) (40000)=80000 from Listing 1.7 as expected.

As a concluding point, we were able to estimate the error in the numerical solution (ncase=2) without any additional computation (beyond ncase=1).

TABLE 1.8 Abbreviated numerical output from Listing 1.8.

```
   t       y1       y2       y3       y4       y5       y6       y7
           e1       e2       e3       e4       e5       e6       e7
   0  0.1072   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000
      0.0000   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000

  40  0.0752   0.0020   0.0153   0.0009   0.0195   0.0011   0.0006
      0.0000   0.0000  -0.0000   0.0000  -0.0000  -0.0000   0.0000
                 .                                  .
                 .                                  .
                 .                                  .
               Output from t = 80 to 1920 removed
                 .                                  .
                 .                                  .
                 .                                  .
1960  0.0000   0.0003   0.0003   0.0000   0.1306   0.0071   0.0223
      0.0000   0.0000  -0.0000   0.0000  -0.0000  -0.0000   0.0000

2000  0.0000   0.0002   0.0002   0.0000   0.1307   0.0071   0.0223
      0.0000   0.0000  -0.0000   0.0000  -0.0000  -0.0000   0.0000

ncall = 80000
```

To conclude this section, we can note that eqs. (1.12) are a particular second-order Runge–Kutta method from a group generally defined by the selection of some arbitrary constants. However, the Euler method of eq. (1.4) is the only (unique) first-order Runge–Kutta method.

1.11 Modified Euler Method as an Embedded Method

Since the Euler and the modified Euler solutions in the preceding section are the same, this suggests that the numerical solution has been confirmed (at least to four figures) by agreement of the solutions from algorithms of two different orders. Also, as the first Runge–Kutta derivative, k_1, is the same for both the methods, this suggests that the first-order (Euler) method is embedded in the second-order (modified Euler) method. The idea that two embedded

algorithms can be used as the basis for estimating the errors in a numerical solution (e.g., as programmed in Listing 1.8 for `ncase=2`) is now considered. Eqs. (1.12) can be written in an alternate form that provides an explicit estimate of the error, ε.

$$k_1 = f(y_i, t_i)h \tag{1.13a}$$

$$y_{i+1}^p = y_i + k_1 \tag{1.13b}$$

$$k_2 = f(y_{i+1}^p, t_{i+1})h = f(y_i + k_1, t_i + h)h \tag{1.13c}$$

$$\varepsilon_{i+1} = \frac{k_2 - k_1}{2} \tag{1.13d}$$

$$y_{i+1}^c = y_{i+1}^p + \varepsilon_{i+1} \tag{1.13e}$$

where the superscripts p and c indicate a predicted value and a corrected value, respectively. Note that

$$y_{i+1}^c = y_i + k_1 + \frac{k_2 - k_1}{2} = y_i + \frac{k_1 + k_2}{2} \tag{1.13f}$$

which is just the modified Euler method.

We can note the following properties of eqs. (1.13).

- Eq. (1.13a) gives the first Runge–Kutta derivative, k_1.
- Eq. (1.13b) is the Euler method (eq. (1.4)), a first-order method based on k_1, that gives the predicted value y_{i+1}^p.
- Eq. (1.13c) gives the second Runge–Kutta derivative, k_2, from the predicted value y_{i+1}^p
- Eq. (1.13d) is an explicit estimate of the solution error, ε, computed from k_1 and k_2.
- Eq. (1.13e) gives a corrected value of the solution, y_{i+1}^c, by adding the estimated error to the predicted value as a correction.
- Eq. (1.13f) confirms that the corrected value is the same as that for the second-order modified Euler method. Thus, we can consider the result of eq. (1.13e) as a first-order method embedded in a second-order method. The key to this idea of an embedded pair is the same Runge–Kutta derivative k_1 for both methods.

The implementation (programming) of eqs. (1.13) is a straightforward modification of the `ncase=2` programming in Listing 1.8.

```
#
# Modified Euler integration
  for(i1 in 2:nout){
#
#   nt modified Euler steps
    for(i2 in 1:nt){
    if(ncase==1){
      yb=y
      rk1=bioreactor_8(t,y)*h
      y1=yb+rk1; t=t+h
      rk2=bioreactor_8(t,y1)*h
      y=yb+(rk1+rk2)/2
      ee=y-y1
    }
    if(ncase==2){
      yb=y
      rk1=bioreactor_8(t,y)*h
      y1=yb+rk1; t=t+h
      rk2=bioreactor_8(t,y1)*h
      ee=(rk2-rk1)/2
      y=y1+ee
    }
    }
```

Listing 1.9: Programming of the Euler and modified Euler methods as an embedded pair.

The ODE routine is `bioreactor_8` and is the same as Listing 1.6b. Eqs. (1.13) is programmed as `ncase=2`. The correspondence of the programming and eqs. (1.13) is clear. Note that the estimated error `ee` computed according to eq. (1.13d), `ee=(rk2-rk1)/2`, is added as a correction, `y=y1+ee`, according to eq. (1.13e). The final left `}` concludes the `for` loop in `i2`.

The explicit error estimate ε_{i+1} can be used to adjust the integration step h according to a specified error tolerance. If ε_{i+1} is above the specified tolerance, h can be reduced and the step from i to $i+1$ is repeated. This reduction in h can be repeated until the estimated error is less than the specified error tolerance, at which point h can

be considered small enough to achieve the required accuracy in the numerical solution (from the Euler method of eq. (1.13b)). The estimated error can then be added as a correction to the (predicted) solution at $i+1$ (eq. (1.13e)) to produce an improved (corrected) value that is the starting value for the next step along the solution, from $i+1$ to $i+2$.

Note that this procedure of using the estimated error to adjust the integration step requires only the ODE (eq. (1.3a)) and not an exact solution of the ODE. However, we should keep in mind that eq. (1.13d) provides an estimate of the error and not the exact error (that would generally require the exact solution). The details of adjusting the integration step h is not considered here. However, step size adjustment to meet a specified error tolerance is used in most quality library integrators such as `lsoda` (in the R utility `ode`) and `lsodes`.

We can now apply these basic ideas to higher order Runge–Kutta methods, including the use of an estimated error computed from an embedded Runge–Kutta pair. In conclusion, this development and implementation (programming) of higher order Runge–Kutta methods is based on the fundamental idea that these methods fit the underlying Taylor series of the ODE solution to any number of terms without having to differentiate the ODE (they require only the first derivative in $\frac{dy}{dt} = f(y,t)$ of eq. (1.3a) at selected points along the solution).

1.12 Classic Fourth-Order Runge–Kutta Method as an Embedded Method

The classic fourth-order Runge–Kutta method, which was reported more than 100 years ago, is

$$k_1 = f(y_i, t_i)h \tag{1.14a}$$

$$k_2 = f(y_i + k_1/2, t_i + h/2)h \tag{1.14b}$$

$$k_3 = f(y_i + k_2/2, t_i + h/2)h \tag{1.14c}$$

$$k_4 = f(y_i + k_3, t_i + h)h \tag{1.14d}$$

$$y_{4,i+1} = y_i + (1/6)(k_1 + 2k_2 + 2k_3 + k_4);\ t_{i+1} = t_i + h \tag{1.14e}$$

Eqs. (1.14) fit the Taylor series up to and including the fourth-order derivative term, $\frac{d^4y}{dt^4}\frac{h^4}{4!}$, that is, the resulting numerical solution is $O(h^4)$. This higher order is achieved by evaluating the Runge–Kutta derivatives k_1, k_2, k_3, and k_4 at the points $t_i, t_i + h/2, t_i + h/2$, and $t_i + h$, respectively.

The second-order Runge–Kutta method can be used in combination with eqs. (1.14).

$$k_1 = f(y_i, t_i)h \tag{1.15a}$$

$$k_2 = f(y_i + k_1/2, t_i + h/2)h \tag{1.15b}$$

$$y_{2,i+1} = y_i + k_2;\ t_{i+1} = t_i + h \tag{1.15c}$$

which is the midpoint method. As the name suggests, k_2 is computed at the midpoint between i and $i + 1$, that is, at $t_i + h/2$.

The second-order midpoint Runge–Kutta method of eqs. (1.15) has the same k_1 and k_2 as the classic fourth-order Runge–Kutta method (compare eqs. (1.14a) and (1.15a), eqs. (1.14b) and (1.15b)), and therefore, this second-order method is embedded in the fourth-order method. An error estimate for this second-order method can be obtained by subtracting the second-order solution $y_{2,i+1}$ (eq. (1.15c)) from the fourth-order solution $y_{4,i+1}$ (eq. (1.14e)).

$$\begin{aligned}\epsilon_{i+1} &= y_{4,i+1} - y_{2,i+1} = y_i + (1/6)(k_1 + 2k_2 + 2k_3 + k_4) - (y_i + k_2) \\ &= (1/6)(k_1 - 4k_2 + 2k_3 + k_4)\end{aligned} \tag{1.16}$$

Note how the k_1 and k_2 terms combine in arriving at eq. (1.16) because they are the same for both algorithms. As this error estimate was achieved by subtracting the second-order solution from the fourth order solution, it actually represents two terms in the Taylor series, $\frac{d^3y}{dt^3}\frac{h^3}{3!}$ and $\frac{d^4y}{dt^4}\frac{h^4}{4!}$, that is, ϵ_i from eq. (1.16) is a two-term error estimate, and therefore, we might expect that it will be more accurate than the one-term error estimate of eq. (1.13d). Experience has indicated this is the case.

The following are the principal conclusions from this discussion of embedded methods:

- The Runge–Kutta derivatives generally can be computed once for both the lower order and the higher order methods of an embedded pair. In other words, the common Runge–Kutta derivatives are the basis for embedded pairs.
- Correction of the lower order solution using the estimated error (the difference between the higher and lower order solutions) gives a substantially improved lower order solution. In other words, the higher order solution is used as the base point for the next step along the solution.
- The estimated error could be used to adjust the integration step h according to a prescribed error tolerance.

Listing 1.9 is an extension of Listing 1.8 for eqs. (1.14) and (1.15).

```
#
# ODE routine
  setwd("c:/R/bme_ode/chap1")
  source("bioreactor_9.R")
#
# Select classical fourth order Runge Kutta method
#
# ncase = 1: RKC4
#
# ncase = 2: RKC4 with embedded second order midpoint
#            method and error estimate
  ncase=2;
#
# Parameter values for BP10001
  k1=8.87e-03;
  k2=13.18;
  k3=0.129;
  k4=0.497;
  k5=0.027;
  k6=0.545e-3;
  km2=87.7;
  km3=99.9;
#
```

```
# Initial condition
  n=7;nout=51;t=0;ncall=0;
  y=c(0.10724,0,0,0,0,0,0)
  cat(sprintf(
  "\n        t      y1      y2      y3      y4      y5
     y6      y7"))
  if(ncase==2){
  ee=c(0,0,0,0,0,0,0)
    cat(sprintf(
  "\n               e1      e2      e3      e4      e5
     e6      e7"))}
  cat(sprintf("\n %8.0f%8.4f%8.4f%8.4f%8.4f%8.4f%8.4f
     %8.4f",
  t,y[1],y[2],y[3],y[4],y[5],y[6],y[7]))
  if(ncase==2){
    cat(sprintf("\n         %8.4f%8.4f%8.4f%8.4f%8.4f%8.4f
       %8.4f\n",
    ee[1],ee[2],ee[3],ee[4],ee[5],ee[6],ee[7]))}
#
# Arrays for output
  out=matrix(0,nrow=nout,ncol=(n+1))
  out[1,-1]=y
  out[1,1]=t
#
# Parameters for t integration
# nt=400;h=0.10  # unstable
  nt=800;h=0.05  # stable
#
# rkc4 integration
  for(i1 in 2:nout){
#
#   nt rkc4 steps
    for(i2 in 1:nt){
    if(ncase==1){
      yb=y; tb=t
      rk1=bioreactor_9(tb,yb)*h
      y=yb+0.5*rk1; t=tb+0.5*h
      rk2=bioreactor_9(t,y)*h
      y=yb+0.5*rk2; t=tb+0.5*h
      rk3=bioreactor_9(t,y)*h
      y=yb+rk3; t=tb+h
```

```
      rk4=bioreactor_9(t,y)*h
      y=yb+(1/6)*(rk1+2*rk2+2*rk3+rk4)
    }
    if(ncase==2){
      yb=y; tb=t
      rk1=bioreactor_9(tb,yb)*h
      y=yb+0.5*rk1; t=tb+0.5*h
      rk2=bioreactor_9(t,y)*h
      y2=yb+rk2
      y=yb+0.5*rk2; t=tb+0.5*h
      rk3=bioreactor_9(t,y)*h
      y=yb+rk3; t=tb+h
      rk4=bioreactor_9(t,y)*h
      y=yb+(1/6)*(rk1+2*rk2+2*rk3+rk4)
      ee=y-y2
    }
    }
#
#   Solution after nt rkc4 steps
    cat(sprintf("\n %8.0f%8.4f%8.4f%8.4f%8.4f%8.4f%8.4f
       %8.4f",
    t,y[1],y[2],y[3],y[4],y[5],y[6],y[7]))
    if(ncase==2){
      cat(sprintf("\n         %8.4f%8.4f%8.4f%8.4f%8.4f
         %8.4f%8.4f\n",
      ee[1],ee[2],ee[3],ee[4],ee[5],ee[6],ee[7]))}
    out[i1,-1]=y
    out[i1,1]=t
  }
#
# Calls to bioreactor_9
  cat(sprintf("\n ncall = %5d\n\n",ncall))
#
# Single plot
  par(mfrow=c(1,1))
#
# y1
  plot(out[,1],out[,2],type="l",xlab="t",ylab="y1(t),...,
     y7(t)",
    xlim=c(0,2000),ylim=c(0,0.14),lty=1, main="y1(t),...,
       y7(t) vs t",
```

```
    lwd=2)
#
# y2
  lines(out[,1],out[,3],type="l",lty=2,lwd=2)
#
# y3
  lines(out[,1],out[,4],type="l",lty=3,lwd=2)
#
# y4
  lines(out[,1],out[,5],type="l",lty=4,lwd=2)
#
# y5
  lines(out[,1],out[,6],type="l",lty=5,lwd=2)
#
# y6
  lines(out[,1],out[,7],type="l",lty=6,lwd=2)
#
# y7
  lines(out[,1],out[,8],type="l",lty=7,lwd=2)
```

Listing 1.10: The classic fourth-order Runge–Kutta with the embedded midpoint method.

We can note the following details about Listing 1.10.

- The ODE routine is `bioreactor_9` and is the same as Listing 1.6b. Also, two cases are programmed. `ncase=1` is for eqs. (1.14). `ncase=2` is for a combination of eqs. (1.14) and (1.15), including the estimated error.

```
#
# ODE routine
  setwd("c:/R/bme_ode/chap1")
  source("bioreactor_9.R")
#
# Select classical fourth order Runge Kutta method
#
# ncase = 1: RKC4
#
# ncase = 2: RKC4 with embedded second order midpoint
```

```
#            method and error estimate
  ncase=2;
```

- The programming of the parameters for eqs. (1.1) is the same as in Listing 1.8.
- The ICs and the output array `out` of Listing 1.8 are repeated.
- The two cases for the ODE integration are programmed as

```
#
# Parameters for t integration
# nt=400;h=0.10  # unstable
  nt=800;h=0.05  # stable
#
# rkc4 integration
  for(i1 in 2:nout){
#
#   nt rkc4 steps
    for(i2 in 1:nt){
    if(ncase==1){
      yb=y; tb=t
      rk1=bioreactor_9(tb,yb)*h
      y=yb+0.5*rk1; t=tb+0.5*h
      rk2=bioreactor_9(t,y)*h
      y=yb+0.5*rk2; t=tb+0.5*h
      rk3=bioreactor_9(t,y)*h
      y=yb+rk3; t=tb+h
      rk4=bioreactor_9(t,y)*h
      y=yb+(1/6)*(rk1+2*rk2+2*rk3+rk4)
    }
    if(ncase==2){
      yb=y; tb=t
      rk1=bioreactor_9(tb,yb)*h
      y=yb+0.5*rk1; t=tb+0.5*h
      rk2=bioreactor_9(t,y)*h
      y2=yb+rk2
      y=yb+0.5*rk2; t=tb+0.5*h
      rk3=bioreactor_9(t,y)*h
      y=yb+rk3; t=tb+h
      rk4=bioreactor_9(t,y)*h
      y=yb+(1/6)*(rk1+2*rk2+2*rk3+rk4)
```

```
  ee=y-y2
}
}
```

The ODE routine is `bioreactor_9` and is the same as Listing 1.6b. For `ncase=1`, the programming is for eqs. (1.14). For `ncase=2`, the error `ee` is estimated as the difference between the fourth-order Runge–Kutta and the second-order midpoint method of eqs. (1.15) (`ee=y-y2`).

- The solution and estimated errors are displayed after `nt` steps (performed in the `for` loop with index `i2`). The final left `}` concludes the `for` loop in `i2`.
- The number of calls to the ODE routine `bioreactor_9` is `ncall=160000` as expected (four derivative evaluations for each Euler step, with 40,000 Euler steps). The numerical and graphical outputs are the same as for Listings 1.10 and 1.11 (including Table 1.6).

Eqs. (1.16) can easily be programmed as a variant of the `ncase=2` code.

```
ee=(1/6)*(rk1-4*rk2+2*rk3+k4)
y=y2+ee
```

Note also that the integration step is again $h = 0.05$. We might expect that the higher order method of eqs. (1.14) would permit a larger step. This is true with respect to accuracy, but it is not true with respect to stability. In fact, the constant c in eq. (1.5) for the fourth-order method of eqs. (1.14) is only 2.785. In other words, the calculation of four derivatives in eqs. (1.14) extended the stability limit of the Euler and modified Euler methods only slightly from $c = 2$ to $c = 2.785$; of course, the advantage of doing the additional derivative calculations is the increase in accuracy from $O(h^2)$ (modified Euler method) to $O(h^4)$ (the fourth-order method of eqs. (1.14)). But the numerical integration of eqs. (1.1) (which are stiff) is limited by stability to a step of $h = 0.05$ (and not limited by accuracy because all of the preceding numerical solutions were similar to four figures);

in other words, we might say the integration step of $h = 0.05$ was excessively small with regard to accuracy.

1.13 RKF45 Method

To conclude the discussion of ODE numerical integration with explicit Runge–Kutta methods, we consider a widely used embedded pair usually designated as `RKF45` [5, p 84].

$$k_1 = f(y_i, t_i)h \tag{1.17a}$$

$$k_2 = f(y_i + k_1/4, t_i + h/4)h \tag{1.17b}$$

$$k_3 = f(y_i + (3/32)k_1 + (9/32)k_2, t_i + (3/8)h)h \tag{1.17c}$$

$$k_4 = f(y_i + (1932/2197)k_1 - (7200/2197)k_2 + (7296/2197)k_3, t_i + (12/13)h)h \tag{1.17d}$$

$$k_5 = f(y_i + (439/216)k_1 - 8k_2 + (3680/513)k_3 - (845/4104)k_4, t_i + h)h \tag{1.17e}$$

$$k_6 = f(y_i - (8/27)k_1 + 2k_2 - (3544/2565)k_3 + (1859/4104)k_4 - (11/40)k_5, t_i + (1/2)h)h \tag{1.17f}$$

An $O(h^4)$ method is then

$$y_{4,i+1} = y_i + (25/216)k_1 + (1408/2565)k_3 + (2197/4104)k_4 - (1/5)k_5 \tag{1.17g}$$

and an $O(h^5)$ method is (with the same ks)

$$y_{5,i+1} = y_i + (16/315)k_1 + (6656/12825)k_3 + (28561/56430)k_4 - (9/50)k_5 + (2/55)k_6 \tag{1.17h}$$

An error estimate can then be obtained by subtracting eq. (1.17g) from eq. (1.17h).

$$\epsilon_{i+1} = y_{i+1,5} - y_{i+1,4} \tag{1.17i}$$

Note that six derivative evaluations are required (k_1 through k_6), even though the final result from eq. (1.17h) is only $O(h^5)$ (the number of derivative evaluations will, in general, be equal to or greater than the order of the method).

The formulas of eqs. (1.17g) and (1.17h) match the Taylor series up to and including the terms $\frac{d^4y_i}{dt^4}\frac{h^4}{4!}$ and $\frac{d^5y_i}{dt^5}\frac{h^5}{5!}$, respectively.

Eqs. (1.17) is implemented in Listing 1.11.

```
#
# ODE routine
  setwd("c:/R/bme_ode/chap1")
  source("bioreactor_10.R")
#
# Select rkf45 format
#
# ncase = 1: No error estimation
#
# ncase = 2: With error estimation
  ncase=2;
#
# Parameter values for BP10001
  k1=8.87e-03;
  k2=13.18;
  k3=0.129;
  k4=0.497;
  k5=0.027;
  k6=0.545e-3;
  km2=87.7;
  km3=99.9;
#
# Initial condition
  n=7;nout=51;t=0;ncall=0;
  y=c(0.10724,0,0,0,0,0,0)
  cat(sprintf(
  "\n        t      y1      y2      y3      y4      y5
     y6      y7"))
  if(ncase==2){
  ee=c(0,0,0,0,0,0,0)
    cat(sprintf(
  "\n               e1      e2      e3      e4      e5
     e6      e7"))}
```

```
  cat(sprintf("\n %8.0f%8.4f%8.4f%8.4f%8.4f%8.4f%8.4f
     %8.4f",
  t,y[1],y[2],y[3],y[4],y[5],y[6],y[7]))
  if(ncase==2){
    cat(sprintf("\n         %8.4f%8.4f%8.4f%8.4f%8.4f%8.4f
       %8.4f\n",
    ee[1],ee[2],ee[3],ee[4],ee[5],ee[6],ee[7]))}
#
# Arrays for output
  out=matrix(0,nrow=nout,ncol=(n+1))
  out[1,-1]=y
  out[1,1]=t
#
# Parameters for t integration
# nt=400;h=0.10  # unstable
  nt=800;h=0.05  # stable
#
# rkf45 integration
  for(i1 in 2:nout){
#
#   nt rkf45 steps
    for(i2 in 1:nt){
    if(ncase==1){
      yb=y; tb=t;
      rk1=bioreactor_10(tb,yb)*h
      y=yb+0.25*rk1;
      t=tb+0.25*h;
      rk2=bioreactor_10(t,y)*h
      y=yb+(3/32)*rk1+(9/32)*rk2;
      t=tb+(3/8)*h;
      rk3=bioreactor_10(t,y)*h
      y=yb+(1932/2197)*rk1-(7200/2197)*rk2+(7296/2197)
         *rk3;
      t=tb+(12/13)*h;
      rk4=bioreactor_10(t,y)*h
      y=yb+(439/216)*rk1-8*rk2 +(3680/513)*rk3 -(845/4104)
         *rk4;
      t=tb+h;
      rk5=bioreactor_10(t,y)*h
      y=yb-(8/27)*rk1+2*rk2-(3544/2565)*rk3+(1859/4104)
         *rk4-(11/40)*rk5;
```

```
      t=tb+0.5*h;
      rk6=bioreactor_10(t,y)*h
      y=yb+(16/135)*rk1+(6656/12825)*rk3+(28561/56430)
         *rk4-(9/50)*rk5+
           (2/55)*rk6;
      t=tb+h;
    }
    if(ncase==2){
      yb=y; tb=t;
      rk1=bioreactor_10(tb,yb)*h
      y=yb+0.25*rk1;
      t=tb+0.25*h;
      rk2=bioreactor_10(t,y)*h
      y=yb+(3/32)*rk1+(9/32)*rk2;
      t=tb+(3/8)*h;
      rk3=bioreactor_10(t,y)*h
      y=yb+(1932/2197)*rk1-(7200/2197)*rk2+(7296/2197)
         *rk3;
      t=tb+(12/13)*h;
      rk4=bioreactor_10(t,y)*h
      y=yb+(439/216)*rk1-8*rk2 +(3680/513)*rk3 -(845/4104)
         *rk4;
      t=tb+h;
      rk5=bioreactor_10(t,y)*h
      y=yb-(8/27)*rk1+2*rk2-(3544/2565)*rk3+(1859/4104)
         *rk4-(11/40)*rk5;
      t=tb+0.5*h;
      rk6=bioreactor_10(t,y)*h
#
#     Fourth order step
      y4=yb+(25/216)*rk1+(1408/2565)*rk3  +(2197/4104)
         *rk4-( 1/5)*rk5;
#
#     Fifth order step
      y=yb+(16/135)*rk1+(6656/12825)*rk3+(28561/56430)
         *rk4-(9/50)*rk5+
           (2/55)*rk6;
      t=tb+h;
#
#     Truncation error estimate
      ee=y-y4
```

```
    }
    }
#
#   Solution after nt rkf45 steps
    cat(sprintf("\n %8.0f%8.4f%8.4f%8.4f%8.4f%8.4f%8.4f
       %8.4f",
    t,y[1],y[2],y[3],y[4],y[5],y[6],y[7]))
    if(ncase==2){
      cat(sprintf("\n         %8.4f%8.4f%8.4f%8.4f%8.4f
         %8.4f%8.4f\n",
      ee[1],ee[2],ee[3],ee[4],ee[5],ee[6],ee[7]))}
    out[i1,-1]=y
    out[i1,1]=t
  }
#
# Calls to bioreactor_10
  cat(sprintf("\n ncall = %5d\n\n",ncall))
#
# Single plot
  par(mfrow=c(1,1))
#
# y1
  plot(out[,1],out[,2],type="l",xlab="t",ylab="y1(t),...,
     y7(t)",
    xlim=c(0,2000),ylim=c(0,0.14),lty=1, main="y1(t),...,
       y7(t) vs t",
    lwd=2)
#
# y2
  lines(out[,1],out[,3],type="l",lty=2,lwd=2)
#
# y3
  lines(out[,1],out[,4],type="l",lty=3,lwd=2)
#
# y4
  lines(out[,1],out[,5],type="l",lty=4,lwd=2)
#
# y5
  lines(out[,1],out[,6],type="l",lty=5,lwd=2)
#
# y6
```

```
  lines(out[,1],out[,7],type="l",lty=6,lwd=2)
#
# y7
  lines(out[,1],out[,8],type="l",lty=7,lwd=2)
```

Listing 1.11: Implementation of the RKF45 method.

We can note the following details of Listing 1.11.

- The ODE routine is `bioreactor_10`, the same as Listing 1.6b. Also, two cases are programmed. `ncase=1` is for just the fifth-order method of eqs. (1.17). `ncase=2` is for a combination of the fourth- and fifth-order methods with an estimated error.

```
#
# ODE routine
  setwd("c:/R/bme_ode/chap1")
  source("bioreactor_10.R")
#
# Select rkf45 format
#
# ncase = 1: No error estimation
#
# ncase = 2: With error estimation
  ncase=2
```

- The programming of the parameters for eqs. (1.1) is the same as in Listing 1.10.
- The ICs, the output array `out`, and the integration parameters of Listing 1.10 are repeated.
- The two cases for the ODE integration are programmed as

```
#
# rkf45 integration
  for(i1 in 2:nout){
#
#   nt rkf45 steps
    for(i2 in 1:nt){
    if(ncase==1){
      yb=y; tb=t;
```

```
    rk1=bioreactor_10(tb,yb)*h
    y=yb+0.25*rk1;
    t=tb+0.25*h;
    rk2=bioreactor_10(t,y)*h
    y=yb+(3/32)*rk1+(9/32)*rk2;
    t=tb+(3/8)*h;
    rk3=bioreactor_10(t,y)*h
    y=yb+(1932/2197)*rk1-(7200/2197)*rk2+(7296/2197)
       *rk3;
    t=tb+(12/13)*h;
    rk4=bioreactor_10(t,y)*h
    y=yb+(439/216)*rk1-8*rk2 +(3680/513)*rk3
       -(845/4104)*rk4;
    t=tb+h;
    rk5=bioreactor_10(t,y)*h
    y=yb-(8/27)*rk1+2*rk2-(3544/2565)*rk3
       +(1859/4104)*rk4-(11/40)*rk5;
    t=tb+0.5*h;
    rk6=bioreactor_10(t,y)*h
    y=yb+(16/135)*rk1+(6656/12825)*rk3+(28561/56430)
       *rk4-(9/50)*rk5+
         (2/55)*rk6;
    t=tb+h;
  }
  if(ncase==2){
    yb=y; tb=t;
    rk1=bioreactor_10(tb,yb)*h
    y=yb+0.25*rk1;
    t=tb+0.25*h;
    rk2=bioreactor_10(t,y)*h
    y=yb+(3/32)*rk1+(9/32)*rk2;
    t=tb+(3/8)*h;
    rk3=bioreactor_10(t,y)*h
    y=yb+(1932/2197)*rk1-(7200/2197)*rk2+(7296/2197)
       *rk3;
    t=tb+(12/13)*h;
    rk4=bioreactor_10(t,y)*h
    y=yb+(439/216)*rk1-8*rk2 +(3680/513)*rk3-(845/
       4104)*rk4;
    t=tb+h;
    rk5=bioreactor_10(t,y)*h
```

```
      y=yb-(8/27)*rk1+2*rk2-(3544/2565)*rk3+(1859/
         4104)*rk4-(11/40)*rk5;
      t=tb+0.5*h;
      rk6=bioreactor_10(t,y)*h
#
#     Fourth order step
      y4=yb+(25/216)*rk1+(1408/2565)*rk3  +(2197/4104)
         *rk4-(1/5)*rk5;
#
#     Fifth order step
      y=yb+(16/135)*rk1+(6656/12825)*rk3+(28561/56430)
         *rk4-(9/50)*rk5+
           (2/55)*rk6;
      t=tb+h;
#
#     Truncation error estimate
      ee=y-y4
    }
    }
```

For `ncase=1`, the programming is for the fifth-order method of eqs. (1.16) (eq. (1.17g) is not used). For `ncase=2`, the error `ee` is estimated as the difference between the fourth-order method (`y4` from eq. (1.17g)) and the fifth- order method (`y` from eq. (1.17h)), that is, `ee=y-y4`.

- The solution and estimated errors are displayed after `nt` steps (performed in the `for` loop with index `i2`). The final left `}` concludes the `for` loop in `i2`.
- The number of calls to the ODE routine `bioreactor_10` is `ncall=240000`. The numerical and graphical outputs are the same as for Listing 1.10. The value of `ncall` is from six derivative evaluations, `(6)(40000)=240000`.

The name RKF45 reflects the Runge–Kutta–Fehlberg method based on a fourth-order method embedded in a fifth-order method.

This concludes the discussion of the numerical integration of eqs. (1.1). The intent was to provide an introduction to the ODE integration in terms of some selected explicit Runge–Kutta methods, that is, the Euler, the modified Euler, the classic fourth order, and the RKF45

methods. An important point to note is the large number of derivative evaluations required to maintain stability, for example, `ncall=240000` for RKF45. In comparison, `lsoda` of `ode` required only 427 derivative evaluations (see Listing 1.7b and the related discussion). Thus, the automatic switching between stiff and nonstiff methods in `lsoda` was very effective.

Generally, an implicit integrator should be used for stiff ODEs. To this end, we consider some low order, fixed step implicit integrators in Appendix A1. However, for nonstiff ODEs, the explicit Runge–Kutta methods discussed previously can be very effective in computing an accurate ODE solution and should therefore be considered. For example, in Chapter 2, a nonstiff ODE model is considered for which explicit algorithms give an accurate solution with fewer derivative evaluations than `lsoda` in `ode`. In addition, explicit methods require fewer calculations at each point along the solution.

These integrators can be programmed as separate, stand-alone routines which would simplify the programming (make it more modular); examples of the use of a separate integrator routine are given in Chapter 2. Also, explicit Runge–Kutta library integrators such as RKF45 are included in `deSolve`. These library integrators have automatic step adjustment in accordance with a user-specified error tolerance (but the source code is not available as in the preceding listings).

References

[1] Beard, D.A. (2012), *Biosimulation: Simulation of Living Systems*, Cambridge University Press, Cambridge, UK.

[2] Butcher, J.W. (2003), *Numerical Methods for Ordinary Differential Equations*, John Wiley & Sons, Inc., Hoboken, NJ.

[3] Butcher, J.W., *Runge–Kutta Methods*, Scholarpedia, vol. 2, no. 9, 2007, p 3147; available at: `http://www.scholarpedia.org/article/Runge-Kutta_methods`.

[4] Deuflhard, P., and F. Bornemann (2002), *Scientific Computing with Ordinary Differential Equations*, Springer-Verlag, New York.

[5] Iserles, A. (1996), *A First Course in the Numerical Analysis of Differential Equations*, Cambridge University Press, Cambridge, UK.

[6] Lee, H.J., and W.E. Schiesser (2004), *Ordinary and Partial Differential Equation Routines, in C, C++, Fortran, Java, Maple and Matlab*, CRC Press, Boca Raton, FL.

[7] Schiesser, W.E. (2013), *Partial Differential Equation Analysis in Biomedical Engineering*, Cambridge University Press, Cambridge, UK.

[8] Shampine, L.F., and S. Thompson (2007a), Stiff systems, Scholarpedia, vol. 2, no. 3, p 2855; available at: `http://www.scholarpedia.org/article/Stiff_systems`.

[9] Shampine, L.F., and S. Thompson (2007b), *Initial value problems*, Scholarpedia, vol. 2, no. 3, p 2861; available at: `http://www.scholarpedia.org/article/Initial_value_problems`.

[10] Soetaert, K., J. Cash, and F. Mazzia (2012), *Solving Differential Equations in R*, Springer-Verlag, Heidelberg, Germany.