Ordinary and Partial Differential Equation Routines in C, C++, Fortran, Java®, Maple®, and MATLAB®

H.J. Lee and W.E. Schiesser
Preface

Initial value ordinary differential equations (ODEs) and partial differential equations (PDEs) are among the most widely used forms of mathematics in science and engineering. However, insights from ODE/PDE-based models are realized only when solutions to the equations are produced with acceptable accuracy and with reasonable effort.

Most ODE/PDE models are complicated enough (e.g., sets of simultaneous nonlinear equations) to preclude analytical methods of solution; instead, numerical methods must be used, which is the central topic of this book.

The calculation of a numerical solution usually requires that well-established numerical integration algorithms are implemented in quality library routines. The library routines in turn can be coded (programmed) in a variety of programming languages. Typically, for a scientist or engineer with an ODE/PDE-based mathematical model, finding routines written in a familiar language can be a demanding requirement, and perhaps even impossible (if such routines do not exist).

The purpose of this book, therefore, is to provide a set of ODE/PDE integration routines written in six widely accepted and used languages. Our intention is to facilitate ODE/PDE-based analysis by using the library routines to compute reliable numerical solutions to the ODE/PDE system of interest.

However, the integration of ODE/PDEs is a large subject, and to keep this discussion to reasonable length, we have limited the selection of algorithms and the associated routines. Specifically, we concentrate on explicit (nonstiff) Runge Kutta (RK) embedded pairs. Within this setting, we have provided integrators that are both fixed step and variable step; the latter accept a user-specified error tolerance and attempt to compute a solution to this required accuracy. The discussion of ODE integration includes truncation error monitoring and control, h and p refinement, stability and stiffness, and explicit and implicit algorithms. Extensions to stiff systems are also discussed and illustrated through an ODE application; however, a detailed presentation of stiff (implicit) algorithms and associated software in six languages was judged impractical for a book of reasonable length.

Further, we have illustrated the application of the ODE integration routines to PDEs through the method of lines (MOL). Briefly, the spatial (boundary value) derivatives of the PDEs are approximated algebraically, typically by finite differences (FDs); the resulting system of initial-value ODEs is then solved numerically by one of the ODE routines.
Thus, we have attempted to provide the reader with a set of computational tools for the convenient solution of ODE/PDE models when programming in any of the six languages. The discussion is introductory with limited mathematical details. Rather, we rely on numerical results to illustrate some basic mathematical properties, and we avoid detailed mathematical analysis (e.g., theorems and proofs), which may not really provide much assistance in the actual calculation of numerical solutions to ODE/PDE problems.

Instead, we have attempted to provide useful computational tools in the form of software. The use of the software is illustrated through a small number of ODE/PDE applications; in each case, the complete code is first presented, and then its components are discussed in detail, with particular reference to the concepts of integration, e.g., stability, error monitoring, and control. Since the algorithms and the associated software have limitations (as do all algorithms and software), we have tried to point out these limitations, and make suggestions for additional methods that could be effective.

Also, we have intentionally avoided using features specific to a particular language, e.g., sparse utilities, object-oriented programming. Rather, we have emphasized the commonality of the programming in the six languages, and thereby illustrate how scientific computation can be done in any of the languages. Of course, language-specific features can be added to the source code that is provided.

We hope this format will allow the reader to understand the basic elements of ODE/PDE integration, and then proceed expeditiously to a numerical solution of the ODE/PDE system of interest. The applications discussed in detail, two in ODEs and two in PDEs, can be used as a starting point (i.e., as templates) for the development of a spectrum of new applications.

We welcome comments and questions about how we might be of assistance (directed to wes1@lehigh.edu). Information for acquiring (gratis) all the source code in this book is available from http://www.lehigh.edu/~wes1/wes1.html. Additional information about the book and software is available from the CRC Press Web site, http://www.crcpress.com.

Dr. Fred Chapman provided expert assistance with the Maple programming. We note with sadness the passing of Jaeson Lee, father of H. J. Lee, during the completion of H. J. Lee’s graduate studies at Lehigh University.

H. J. Lee
W. E. Schiesser
Bethlehem, PA
Contents

1 Some Basics of ODE Integration
   1.1 General Initial Value ODE Problem
   1.2 Origin of ODE Integrators in the Taylor Series
   1.3 The Runge Kutta Method
   1.4 Accuracy of RK Methods
   1.5 Embedded RK Algorithms
   1.6 Library ODE Integrators
   1.7 Stability of RK Methods

2 Solution of a 1x1 ODE System
   2.1 Programming in MATLAB
   2.2 Programming in C
   2.3 Programming in C++
   2.4 Programming in Fortran
   2.5 Programming in Java
   2.6 Programming in Maple

3 Solution of a 2x2 ODE System
   3.1 Programming in MATLAB
   3.2 Programming in C
   3.3 Programming in C++
   3.4 Programming in Fortran
   3.5 Programming in Java
   3.6 Programming in Maple

4 Solution of a Linear PDE
   4.1 Programming in MATLAB
   4.2 Programming in C
   4.3 Programming in C++
   4.4 Programming in Fortran
   4.5 Programming in Java
   4.6 Programming in Maple

5 Solution of a Nonlinear PDE
   5.1 Programming in MATLAB
   5.2 Programming in C
   5.3 Programming in C++
   5.4 Programming in Fortran
5.5 Programming in Java
5.6 Programming in Maple

Appendix A Embedded Runge Kutta Pairs
Appendix B Integrals from ODEs

Appendix C Stiff ODE Integration
   C.1 The BDF Formulas Applied to the 2x2 ODE System
   C.2 MATLAB Program for the Solution of the 2x2 ODE System
   C.3 MATLAB Program for the Solution of the 2x2 ODE System Using ode23s and ode15s

Appendix D Alternative Forms of ODEs
Appendix E Spatial p Refinement
Appendix F Testing ODE/PDE Codes
1

Some Basics of ODE Integration

The central topic of this book is the programming and use of a set of library routines for the numerical solution (integration) of systems of initial value ordinary differential equations (ODEs). We start by reviewing some of the basic concepts of ODEs, including methods of integration, that are the mathematical foundation for an understanding of the ODE integration routines.

1.1 General Initial Value ODE Problem

The general problem for a single initial-value ODE is simply stated as

\[
\frac{dy}{dt} = f(y, t), \quad y(t_0) = y_0 \tag{1.1}(1.2)
\]

where

- \( y \) = dependent variable
- \( t \) = independent variable
- \( f(y, t) \) = derivative function
- \( t_0 \) = initial value of the independent variable
- \( y_0 \) = initial value of the dependent variable

Equations 1.1 and 1.2 will be termed a 1x1 problem (one equation in one unknown). The solution of this 1x1 problem is the dependent variable as a function of the independent variable, \( y(t) \) (this function when substituted into Equations 1.1 and 1.2 satisfies these equations). This solution may be a mathematical function, termed an analytical solution.
To illustrate these ideas, we consider the 1x1 problem, from Braun\(^1\) (which will be discussed subsequently in more detail)

\[
dy{dt} = \lambda e^{-\alpha t} y, \quad y(t_0) = y_0
\]  

(1.3)(1.4)

where \(\lambda\) and \(\alpha\) are positive constants.

Equation 1.3 is termed a *first-order, linear, ordinary differential equation with variable coefficients*. These terms are explained below.

<table>
<thead>
<tr>
<th>Term</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Differential equation</td>
<td>Equation 1.3 has a derivative (dy/dt = f(y, t) = \lambda e^{-\alpha t} y)</td>
</tr>
<tr>
<td>Ordinary</td>
<td>Equation 1.3 has only one independent variable, (t), so that the derivative (dy/dt) is a total or ordinary derivative</td>
</tr>
<tr>
<td>First-order</td>
<td>The highest-order derivative is first order ((dy/dt) is first order)</td>
</tr>
<tr>
<td>Linear</td>
<td>(y) and its derivative (dy/dt) are to the first power; thus, Equation 1.3 is also termed first degree (do not confuse order and degree)</td>
</tr>
<tr>
<td>Variable coefficient</td>
<td>The coefficient (e^{-\alpha t}) is a function of the independent variable, (t) (if it were a function of the dependent variable, (y), Equation 1.3 would be nonlinear or not first degree)</td>
</tr>
</tbody>
</table>

The analytical solution to Equations 1.3 and 1.4 is from Braun:\(^1\)

\[
y(t) = y_0 \exp\left(\frac{\lambda}{\alpha} (1 - \exp(\alpha t))\right), \quad y(0) = y_0
\]  

(1.5)

where \(\exp(x) = e^x\). Equation 1.5 is easily verified as the solution to Equations 1.3 and 1.4 by substitution in these equations:

<table>
<thead>
<tr>
<th>Terms in Equations 1.3 and 1.4</th>
<th>Substitution of Equation 1.5 in Equations 1.3 and 1.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\frac{dy}{dt})</td>
<td>(y_0 \exp\left(\frac{\lambda}{\alpha} (1 - \exp(-\alpha t))\right) \frac{\lambda}{\alpha} \exp(-\alpha t))</td>
</tr>
<tr>
<td></td>
<td>(= \lambda y_0 \exp\left(\frac{\lambda}{\alpha} (1 - \exp(-\alpha t))\right) \exp(-\alpha t))</td>
</tr>
<tr>
<td>(-\lambda e^{-\alpha t} y)</td>
<td>(-\lambda e^{-\alpha t} y_0 \exp\left(\frac{\lambda}{\alpha} (1 - \exp(-\alpha t))\right))</td>
</tr>
<tr>
<td></td>
<td>(= 0)</td>
</tr>
<tr>
<td>(y(0))</td>
<td>(y_0 \exp\left(\frac{\lambda}{\alpha} (1 - \exp(-\alpha(0)))\right) = y_0(e^0) = y_0)</td>
</tr>
</tbody>
</table>

thus confirming Equation 1.5 satisfies Equations 1.3 and 1.4.

Copyright 2004 by Chapman & Hall/CRC
As an example of a \( nxn \) problem (\( n \) ODEs in \( n \) unknowns), we will also subsequently consider in detail the 2x2 system

\[
\frac{dy_1}{dt} = a_{11}y_1 + a_{12}y_2 \quad y_1(0) = y_{10} \\
\frac{dy_2}{dt} = a_{21}y_1 + a_{22}y_2 \quad y_2(0) = y_{20}
\]  

(1.6)

The solution to Equations 1.6 is again the dependent variables, \( y_1, y_2 \), as a function of the independent variable, \( t \). Since Equations 1.6 are linear, constant coefficient ODEs, their solution is easily derived, e.g., by assuming exponential functions in \( t \) or by the Laplace transform. If we assume exponential functions

\[
y_1(t) = c_1e^{\lambda t} \\
y_2(t) = c_2e^{\lambda t}
\]  

(1.7)

where \( c_1, c_2, \) and \( \lambda \) are constants to be determined, substitution of Equations 1.7 in Equations 1.6 gives

\[
c_1\lambda e^{\lambda t} = a_{11}c_1e^{\lambda t} + a_{12}c_2e^{\lambda t} \\
c_2\lambda e^{\lambda t} = a_{21}c_1e^{\lambda t} + a_{22}c_2e^{\lambda t}
\]

Cancellation of \( e^{\lambda t} \) gives a system of algebraic equations (this is the reason assuming exponential solutions works in the case of linear, constant coefficient ODEs)

\[
c_1\lambda = a_{11}c_1 + a_{12}c_2 \\
c_2\lambda = a_{21}c_1 + a_{22}c_2
\]

or

\[
(a_{11} - \lambda)c_1 + a_{12}c_2 = 0 \\
a_{21}c_1 + (a_{22} - \lambda)c_2 = 0
\]  

(1.8)

Equations 1.8 are the 2x2 case of the linear algebraic eigenvalue problem

\[(A - \lambda I)c = 0\]  

(1.9)

where

\[
A = \begin{bmatrix}
a_{11} & a_{12} & \cdots & a_{1n} \\
a_{21} & a_{22} & \cdots & a_{2n} \\
\vdots & \vdots & \ddots & \vdots \\
a_{n1} & a_{n2} & \cdots & a_{nn}
\end{bmatrix}
\]

\[
I = \begin{bmatrix}
1 & 0 & \cdots & 0 \\
0 & 1 & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
0 & 0 & \cdots & 1
\end{bmatrix}
\]
\[
\begin{bmatrix}
    c_1 \\
    c_2 \\
    \vdots \\
    c_n
\end{bmatrix}
\]

\[
0 = \begin{bmatrix}
    0 \\
    0 \\
    \vdots \\
    0
\end{bmatrix}
\]

\(n = 2\) for Equations 1.8, and we use a bold faced symbol for a matrix or a vector.

The preceding matrices and vectors are:

\[A \quad \text{n} \times \text{n} \text{ coefficient matrix}\]
\[I \quad \text{n} \times \text{n} \text{ identity matrix}\]
\[c \quad \text{n} \times 1 \text{ solution vector}\]
\[0 \quad \text{n} \times 1 \text{ zero vector}\]

The reader should confirm that the matrices and vectors in Equation 1.9 have the correct dimensions for all of the indicated operations (e.g., matrix additions, matrix-vector multiples).

Note that Equation 1.9 is a linear, homogeneous algebraic system (homogeneous means that the right-hand side (RHS) is the zero vector). Thus, Equation 1.9, or its 2x2 counterpart, Equations 1.8, will have nontrivial solutions \((c \neq 0)\) if and only if (iff) the determinant of the coefficient matrix is zero, i.e.,

\[|A - \lambda I| = 0\]  \hfill (1.10)

Equation 1.10 is the characteristic equation for Equation 1.9 (note that it is a scalar equation). The values of \(\lambda\) that satisfy Equation 1.10 are the eigenvalues of Equation 1.9. For the 2x2 problem of Equations 1.8, Equation 1.10 is

\[
\begin{vmatrix}
    a_{11} - \lambda & a_{12} \\
    a_{21} & a_{22} - \lambda
\end{vmatrix} = 0
\]

or

\[(a_{11} - \lambda)(a_{22} - \lambda) - a_{21}a_{12} = 0\]  \hfill (1.11)

Equation 1.11 is the characteristic equation or characteristic polynomial for Equations 1.8; note that since Equations 1.8 are a 2x2 linear homogeneous algebraic system, the characteristic equation (Equation 1.11) is a second-order
polynomial. Similarly, since Equation 1.9 is a \( n \times n \) linear homogeneous algebraic system, its characteristic equation is a \( n \)th-order polynomial.

Equation 1.11 can be factored by the quadratic formula

\[
\lambda^2 - (a_{11} + a_{22})\lambda + a_{11}a_{22} - a_{21}a_{12} = 0
\]

\[
\lambda_1, \lambda_2 = \frac{(a_{11} + a_{22}) \pm \sqrt{(a_{11} + a_{22})^2 - 4(a_{11}a_{22} - a_{21}a_{12})}}{2} \tag{1.12}
\]

Thus, as expected, the 2x2 system of Equations 1.8 has two eigenvalues. In general, the \( n \times n \) algebraic system, Equation 1.9, will have \( n \) eigenvalues, \( \lambda_1, \lambda_2, \ldots, \lambda_n \) (which may be real or complex conjugates, distinct or repeated).

Since Equations 1.6 are linear constant coefficient ODEs, their general solution will be a linear combination of exponential functions, one for each eigenvalue

\[
y_1 = c_{11}e^{\lambda_1 t} + c_{12}e^{\lambda_2 t}
\]

\[
y_2 = c_{21}e^{\lambda_1 t} + c_{22}e^{\lambda_2 t} \tag{1.13}
\]

Equations 1.13 have four constants which occur in pairs, one pair for each eigenvalue. Thus, the pair \([c_{11}, c_{21}]^T\) is the eigenvector for eigenvalue \( \lambda_1 \) while \([c_{12}, c_{22}]^T\) is the eigenvector for eigenvalue \( \lambda_2 \). In general, the \( n \times n \) system of Equation 1.9 will have a \( n \times 1 \) eigenvector for each of its \( n \) eigenvalues. Note that the naming convention for any constant in an eigenvector, \( c_{ij} \), is the \( i \)th constant for the \( j \)th eigenvalue. We can restate the two eigenvectors for Equation 1.13 (or Equations 1.8) as

\[
\begin{bmatrix}
  c_{11} \\
  c_{21}
\end{bmatrix}
\begin{bmatrix}
  \lambda_1 \\
  \lambda_2
\end{bmatrix}
\begin{bmatrix}
  c_{12} \\
  c_{22}
\end{bmatrix} \tag{1.14}
\]

Finally, the four constants in eigenvectors (Equations 1.14) are related through the initial conditions of Equations 1.6 and either of Equations 1.8

\[
y_{10} = c_{11}e^{\lambda_1 0} + c_{12}e^{\lambda_2 0}
\]

\[
y_{20} = c_{21}e^{\lambda_1 0} + c_{22}e^{\lambda_2 0} \tag{1.15}
\]

To simplify the analysis somewhat, we consider the special case \( a_{11} = a_{22} = -a, a_{21} = a_{12} = b \), where \( a \) and \( b \) are constants. Then from Equation 1.12,

\[
\lambda_1, \lambda_2 = \frac{-2a \pm \sqrt{(2a)^2 - 4(a^2 - b^2)}}{2} = -a \pm b = -(a - b), -(a + b) \tag{1.16}
\]
From the first of Equations 1.8 for $\lambda = \lambda_1$

$$ (a_{11} - \lambda_1)c_{11} + a_{12}c_{21} = 0 $$

or

$$ (-a + (a - b))c_{11} + bc_{21} = 0 $$

$$ c_{11} = c_{21} $$

Similarly, for $\lambda = \lambda_2$

$$ (a_{11} - \lambda_2)c_{12} + a_{12}c_{22} = 0 $$

or

$$ (-a + (a + b))c_{12} + bc_{22} = 0 $$

$$ c_{12} = -c_{22} $$

Substitution of these results in Equations 1.15 gives

$$ y_{10} = c_{11} - c_{22} $$

$$ y_{20} = c_{11} + c_{22} $$

or

$$ c_{11} = \frac{y_{10} + y_{20}}{2} = c_{21} $$

$$ c_{22} = \frac{y_{20} - y_{10}}{2} = -c_{12} $$

Finally, the solution from Equations 1.13 is

$$ y_1 = \frac{y_{10} + y_{20}}{2} e^{\lambda_1 t} - \frac{y_{20} - y_{10}}{2} e^{\lambda_2 t} $$

$$ y_2 = \frac{y_{10} + y_{20}}{2} e^{\lambda_1 t} + \frac{y_{20} - y_{10}}{2} e^{\lambda_2 t} $$

Equations 1.17 can easily be checked by substitution in Equations 1.6 (with $a_{11} = a_{22} = -a$, $a_{21} = a_{12} = b$) and application of the initial conditions at $t = 0$: 
\[
\frac{dy_1}{dt} = \lambda_1 \frac{y_{10} + y_{20}}{2} e^{\lambda_1 t} - \lambda_2 \frac{y_{20} - y_{10}}{2} e^{\lambda_2 t}
\]
\[
+ a y_1
\]
\[
-b y_2
\]
\[
\frac{dy_2}{dt} = \lambda_1 \frac{y_{10} + y_{20}}{2} e^{\lambda_1 t} + \lambda_2 \frac{y_{20} - y_{10}}{2} e^{\lambda_2 t}
\]
\[
+ a y_2
\]
\[
-b y_1
\]
\[
= 0
\]
\[
= 0
\]

For the initial conditions of Equations 1.6
\[
y_{10} = \frac{y_{10} + y_{20}}{2} e^{\lambda_1 t} - \frac{y_{20} - y_{10}}{2} e^{\lambda_2 t} = y_{10}
\]
\[
y_{20} = \frac{y_{10} + y_{20}}{2} e^{\lambda_1 t} + \frac{y_{20} - y_{10}}{2} e^{\lambda_2 t} = y_{20}
\]
as required.

The ODE problems of Equations 1.3, 1.4, and 1.6 along with their analytical solutions, Equations 1.5 and 1.17, will be used subsequently to demonstrate the use of the ODE integration routines and to evaluate the computed solutions. Since these problems are quite modest (1x1 and 2x2, respectively), we will also subsequently consider two problems with substantially more ODEs. At the same time, these ODE systems will be considered as approximations to partial differential equations (PDEs); in other words, we will use systems of ODEs for the solution of PDEs.

### 1.2 Origin of ODE Integrators in the Taylor Series

In contrast to the analytical solutions presented previously (Equations 1.5 and 1.17), the numerical solutions we will compute are ordered pairs of numbers. For example, in the case of Equation 1.3, we start from the pair \((t_0, y_0)\) (the initial condition of Equation 1.3) and proceed to compute paired values \((t_i, y_i)\) where \(i = 1, 2, \ldots\) is an index indicating a position or point along the solution.
The numerical integration is then a step-by-step algorithm going from the solution point \((t_i, y_i)\) to the point \((t_{i+1}, y_{i+1})\).

This stepping procedure is illustrated in Figure 1.1 and can be represented mathematically by a Taylor series:

\[
y_{i+1} = y_i + \frac{dy_i}{dt} h + \frac{d^2 y_i}{dt^2} \frac{h^2}{2!} + \cdots \tag{1.18}
\]

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

\[
y_{i+1} \approx y_i + \frac{dy_i}{dt} h \tag{1.19}
\]

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. This is the famous Euler’s method.

This stepping procedure is illustrated in Figure 1.1 (with \(i = 0\)). Note that Equation 1.19 is equivalent to projecting along a tangent line from \(i\) to \(i + 1\). In other words, we are representing the solution, \(y(t)\), by a linear approximation. As indicated in Figure 1.1, an error, \(\varepsilon_i\), will occur, which in the case of Figure 1.1 appears to be excessive. However, this apparently large error is only for purposes of illustration in Figure 1.1. By taking a small enough step, \(h\), the error can, at least in principle, be reduced to any acceptable level. To see this, consider the difference between the exact solution, \(y(t_{i+1})\), and the approximate solution, \(y_{i+1}\), if \(h\) is halved in Figure 1.1 (note how the vertical difference corresponding to \(\varepsilon_i\) is reduced). In fact, a major part of this book is devoted to controlling the error, \(\varepsilon_i\), to an acceptable level by varying \(h\). \(\varepsilon_i\) is termed the truncation error, which is a logical name since it results from truncating a
Taylor series (Equation 1.18), in this case, to Equation 1.19. In other words, \( \varepsilon_i \) is the truncation error for Euler’s method, Equation 1.19.

We could logically argue that the truncation error could be reduced (for a given \( h \)) by including more terms in the Taylor, e.g., the second derivative term \((d^2y_i/dt^2)(h^2/2!)\). Although this is technically true, there is a practical problem. For the general ODE, Equation 1.1, we have only the first derivative available

\[
\frac{dy}{dt} = f(y_i, t_i)
\]

The question then in using the second derivative term of the Taylor series is “How do we obtain the second derivative, \( d^2y_i/dt^2 \)?”. One answer would be to differentiate the ODE, i.e.,

\[
\frac{d^2y}{dt^2} = \frac{d}{dt} \left( \frac{dy}{dt} \right) = \frac{\partial f}{\partial y} \frac{dy}{dt} + \frac{\partial f}{\partial t} = \frac{\partial f}{\partial y} f + \frac{\partial f}{\partial t} \tag{1.20}
\]

Then we can substitute Equation 1.20 in Equation 1.18:

\[
y_{i+1} = y_i + f_i h + \left( \frac{\partial f}{\partial y} f + \frac{\partial f}{\partial t} \right) \frac{h^2}{2!} \tag{1.21}
\]

where again subscript “\( i \)” means evaluated at point \( i \).

As an example of the application of Equation 1.21, consider the model ODE

\[
\frac{dy}{dt} = f(y, t) = \lambda y \tag{1.22}
\]

where \( \lambda \) is a constant. Then

\[
f_i = \lambda y_i
\]

\[
\left( \frac{\partial f}{\partial y} f + \frac{\partial f}{\partial t} \right) = \lambda (\lambda y_i)
\]

(note: \( \partial f / \partial t = 0 \) since \( f = \lambda y \) does not depend on \( t \)) and substitution in Equation 1.21 gives

\[
y_{i+1} = y_i + \lambda y_i h + \lambda (\lambda y_i) \frac{h^2}{2!} = y_i (1 + \lambda h + (\lambda h)^2/2!)
\]

\( y_i (1 + \lambda h + (\lambda h)^2/2!) \) is the Taylor series of \( ye^{\lambda h} \) up to and including the \( h^2 \) term, but \( ye^{\lambda h} \) is the analytical solution to Equation 1.22 with the initial condition \( y(t_i) = y_i \) for the integration step, \( h = t_{i+1} - t_i \). Thus, as stated previously, Equation 1.21 fits the Taylor series of the analytical solution to Equation 1.22 up to and including the \((d^2y_i/dt^2)(h^2/2!)\) term.

Of course, we could, in principle, continue this process of including additional terms in the Taylor series, e.g., using the derivative of the second
derivative to arrive at the third derivative, etc. Clearly, however, the method quickly becomes cumbersome (and this is for only one ODE, Equation 1.1). Application of this *Taylor series method* to systems of ODEs involves a lot of differentiation. (Would we want to apply it to a system of 100 or 1000 ODEs? We think not.)

Ideally, we would like to have a higher-order ODE integration method (higher than the first-order Euler method) without having to take derivatives of the ODEs. Although this may seem like an impossibility, it can in fact be done by the *Runge Kutta (RK) method*. In other words, the RK method can be used to fit the numerical ODE solution exactly to an arbitrary number of terms in the underlying Taylor series without having to differentiate the ODE. We will investigate the RK method, which is the basis for the ODE integration routines described in this book.

The other important characteristic of a numerical integration algorithm (in addition to not having to differentiate the ODE) is a way of estimating the truncation error, ε, so that the integration step, h, can be adjusted to achieve a solution with a prescribed accuracy. This may also seem like an impossibility since it would appear that in order to compute ε we need to know the exact (analytical) solution. But if the exact solution is known, there would be no need to calculate the numerical solution. The answer to this apparent contradiction is the fact that we will calculate an estimate of the truncation error (and not the exact truncation error which would imply that we know the exact solution). To see how this might be done, consider computing a truncation error estimate for the Euler method. Again, we return to the Taylor series (which is the mathematical tool for most of the numerical analysis of ODE integration).

Now we will expand the first derivative \( \frac{dy}{dt} \)

\[
\frac{dy_{i+1}}{dt} = \frac{dy_i}{dt} + \frac{d^2y}{dt^2} \frac{h}{1!} + \cdots \tag{1.23}
\]

\( d^2y/dt^2 \) is the second derivative we require in Equation 1.18. If the Taylor series in Equation 1.23 is truncated after the \( h \) term, we can solve for this second derivative

\[
\frac{d^2y_i}{dt^2} = \frac{dy_{i+1}}{dt} - \frac{dy_i}{h} \tag{1.24}
\]

Equation 1.24 seems logical, i.e., the second derivative is a *finite difference (FD) approximation* of the first derivative. Note that Equation 1.24 has the important property that we can compute the second derivative without having to differentiate the ODE; rather, all we have to do is use the ODE twice, at grid points \( i \) and \( i + 1 \). Thus, the previous differentiation of Equation 1.20 is avoided. However, note also that Equation 1.24 gives only an approximation for the second derivative since it results from truncating the Taylor series of Equation 1.23. Fortunately, the approximation of Equation 1.24 will generally
become increasingly accurate with decreasing $h$ since the higher terms in $h$ in Equation 1.23 (after the point of truncation) will become increasingly smaller.

Substituting Equation 1.24 in Equation 1.18 (truncated after the $h^2$ term) gives

$$y_{i+1} = y_i + \frac{dy_i}{dt} h + \frac{d^2 y_i}{dt^2} h^2 + \frac{d^3 y_i}{dt^3} \frac{h^3}{3!}$$

$$= y_i + \frac{dy_i}{dt} h + \frac{d^2 y_i}{dt^2} \frac{h^2}{2!}$$

$$= y_i + \frac{dy_i}{dt} h + \left( \frac{d^2 y_{i+1}}{dt^2} - \frac{d^2 y_i}{dt^2} \right) \frac{h}{2!}$$

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

Equation 1.25 is the well-known modified Euler method or extended Euler method. We would logically expect that for a given $h$, Equation 1.25 will give a more accurate numerical solution for the ODE than Equation 1.19. We will later demonstrate that this is so in terms of some ODE examples, and we will state more precisely how the truncation errors of Equations 1.19 and 1.25 vary with $h$.

Note that Equation 1.25 uses the derivative $dy/dt$ averaged at points $i$ and $i+1$, as illustrated in Figure 1.2. Thus, whereas the derivative at $i$ in Figure 1.1

**Stepping formulas:**

$$y_{i+1}^p = y_i + (dy_i / dt) h$$

$$y_{i+1}^c = y_i + \frac{(dy_i / dt) + (dy_{i+1} / dt)}{2} h$$

$\varepsilon_{i+1}^p, \varepsilon_{i+1}^c$: Truncation errors

**FIGURE 1.2**

Modified Euler method.
is too large and causes the large overshoot of the numerical solution above the exact solution (and thus, a relatively large value of \( \varepsilon_i \)), the averaging of the derivatives at \( i \) and \( i + 1 \) in Figure 1.2 reduces this overshoot (and the truncation error is reduced from \( \varepsilon_i^p \) to \( \varepsilon_i^c \)).

Equation 1.25 can be rearranged into a more useful form. If we assume that the truncation error of Euler’s method, \( \varepsilon_i \), is due mainly to the second derivative term \( (d^2y_i/dt^2)(h^2/2!) \) of Equation 1.18 (which will be the case if the higher-order terms in Equation 1.18 are negligibly small), then

\[
\varepsilon_i = \frac{d^2y_i}{dt^2} \frac{h^2}{2!} = \frac{dy_{i+1}}{dt} - \frac{dy_i}{dt} \frac{h^2}{2!} = \left( \frac{dy_{i+1}}{dt} - \frac{dy_i}{dt} \right) \frac{h}{2!}
\]

and Equation 1.25 can be written as a two-step algorithm:

\[
y_{i+1}^p = y_i + \frac{dy_i}{dt} h
\]

\[
\varepsilon_i = \left( \frac{dy_{i+1}^p}{dt} - \frac{dy_i}{dt} \right) \frac{h}{2!}
\]

\[
y_{i+1}^c = y_i + \frac{dy_i}{dt} h + \varepsilon_i = y_{i+1} + \varepsilon_i
\]

With a little algebra, we can easily show that \( y_{i+1} \) of Equation 1.25 and \( y_{i+1}^p \) of Equation 1.26c are the same. While Equation 1.25 and Equations 1.26c are mathematically equivalent, Equations 1.26 have an advantage when used in a computer program. Specifically, an algorithm that automatically adjusts \( h \) to achieve a prescribed accuracy, \( tol \), can be programmed in the following steps:

1. Compute \( y_{i+1}^p \) by the Euler method, Equation 1.26a. The superscript \( p \) in this case denotes a predicted value.
2. Compute the estimated error, \( \varepsilon_i \), from Equation 1.26b. Note that \( dy_{i+1}^p/dt = f(y_{i+1}^p, t_i + 1) \), where \( t_i + 1 = t_i + h \).
3. Pose the question is \( \varepsilon_i < tol \)? If no, reduce \( h \) and return to 1. If yes, continue to 4.
4. Add \( \varepsilon_i \) from 3 to \( y_{i+1}^p \) to obtain \( y_{i+1}^c \) according to Equation 1.26c. The superscript \( c \) denotes a corrected value.
5. Increment \( i \), advance \( t_i \) to \( t_{i+1} \) by adding \( h \), go to 1. to take the next step along the solution.

The algorithm of Equations 1.26 is termed a predictor-corrector method, which we will subsequently discuss in terms of a computer program.

To conclude this introductory discussion of integration algorithms, we introduce the RK notation. If we define \( k_1 \) and \( k_2 \) (termed Runge Kutta constants...
although they are not constant, but rather, vary along the solution) as

\[ k_1 = f(y_i, t_i)h \]  \hfill (1.27a)
\[ k_2 = f(y_i + k_1, t_i + h)h \]  \hfill (1.27b)

the Euler method of Equation 1.19 can be written as (keep in mind \( dy/dt = f(y, t) \))

\[ y_{i+1} = y_i + k_1 \]  \hfill (1.28)

and the modified Euler method of Equation 1.25 can be written

\[ y_{i+1} = y_i + \frac{k_1 + k_2}{2} \]  \hfill (1.29)

(the reader should confirm that Equation 1.25 and Equation 1.29 are the same).

Also, the modified Euler method written in terms of an explicit error estimate, Equations 1.26, can be conveniently written in RK notation:

\[ y_{i+1}^p = y_i + k_1 \]  \hfill (1.30a)
\[ \epsilon_i = \frac{(k_2 - k_1)}{2} \]  \hfill (1.30b)
\[ y_{i+1}^c = y_i + k_1 + \frac{(k_2 - k_1)}{2} = y_i + \frac{(k_1 + k_2)}{2} \]  \hfill (1.30c)

However, the RK method is much more than just a convenient system of notation. As stated earlier, it is a method for fitting the underlying Taylor series of the ODE solution to any number of terms without having to differentiate the ODE (it requires only the first derivative in \( dy/dt = f(y, t) \) as we observe in Equations 1.29 and 1.30). We next explore this important feature of the RK method, which is the mathematical foundation of the ODE integrators to be discussed subsequently.

### 1.3 The Runge Kutta Method

The RK method consists of a series of algorithms of increasing order. There is only one first order RK method, the Euler method, which fits the underlying Taylor series of the solution up to and including the first derivative term, as indicated by Equation 1.19.

The second-order RK method is actually a family of second-order methods; a particular member of this family is selected by choosing an arbitrary constant in the general second-order RK formulas. The origin of these formulas is
illustrated by the following development (based on the idea that the second-order RK method fits the Taylor series up to and including the second derivative term, \((d^2 y/dt^2)(h^2/2!))\).

We start the analysis with a general RK stepping formula of the form

\[
y_{i+1} = y_i + c_1k_1 + c_2k_2
\]  
(1.31a)

where \(k_1\) and \(k_2\) are RK “constants” of the form

\[
k_1 = f(y_i, t_i)h
\]  
(1.31b)

\[
k_2 = f(y_i + a_2k_1(y_i, t_i), t_i + a_2h)h = f(y_i + a_2 f(y_i, t_i)h, t_i + a_2h)h
\]  
(1.31c)

and \(c_1, c_2\) and \(a_2\) are constants to be determined.

If \(k_2\) from Equation 1.31c is expanded in a Taylor series in two variables,

\[
k_2 = f(y_i + a_2 f(y_i, t_i)h, t_i + a_2h)h
\]

\[
= [f(y_i, t_i) + f_y(y_i, t_i)a_2 f(y_i, t_i)h + f_t(y_i, t_i)a_2h] h + O(h^3)
\]  
(1.32)

Substituting Equations 1.31b and 1.32 in Equation 1.31a gives

\[
y_{i+1} = y_i + c_1 f(y_i, t_i)h + c_2[f(y_i, t_i) + f_y(y_i, t_i)a_2 f(y_i, t_i)h \\
+ f_t(y_i, t_i)a_2h] h + O(h^3)
\]

\[
= y_i + (c_1 + c_2) f(y_i, t_i)h + c_2[f(y_i, t_i)a_2 f(y_i, t_i) \\
+ f_t(y_i, t_i)a_2h] h^2 + O(h^3)
\]  
(1.33)

Note that Equation 1.33 is a polynomial in increasing powers of \(h\); i.e., it has the form of a Taylor series. Thus, if we expand \(y_{i+1}\) in a Taylor series around \(y_i\), we will obtain a polynomial of the same form, i.e., in increasing powers of \(h\)

\[
y_{i+1} = y_i + \frac{dy_i}{dt} h + \frac{d^2 y_i}{dt^2} \frac{h^2}{2!} + O(h^3)
\]

\[
= y_i + f(y_i, t_i)h + \frac{df(y_i, t_i)}{dt} \frac{h^2}{2!} + O(h^3)
\]  
(1.34)

where we have used \(dy_i/dt = f(y_i, t_i)\), i.e., the ODE we wish to integrate numerically. To match Equations 1.33 and 1.34, term-by-term (with like powers of \(h\)), we need to have \([df(y_i, t_i)/dt](h^2/2!))\) in Equation 1.34 in the form of \(f_y(y_i, t_i)a_2 f(y_i, t_i) + f_t(y_i, t_i)a_2\) in Equation 1.33.

If chain-rule differentiation is applied to \(d f(y_i, t_i)/dt\)

\[
\frac{df(y_i, t_i)}{dt} = f_y(y_i, t_i) \frac{dy_i}{dt} + f_t(y_i, t_i) = f_y(y_i, t_i) f(y_i, t_i) + f_t(y_i, t_i)
\]  
(1.35)
Substitution of Equation 1.35 in Equation 1.34 gives

\[ y_{i+1} = y_i + f(y_i, t_i)h + \left[ f_y(y_i, t_i) f(y_i, t_i) + f_t(y_i, t_i) \right] \frac{h^2}{2} + O(h^3) \]  \hspace{1cm} (1.36)

We can now equate coefficients of like powers of \( h \) in Equations 1.33 and 1.36

<table>
<thead>
<tr>
<th>Power of ( h )</th>
<th>Equation 1.33</th>
<th>Equation 1.36</th>
</tr>
</thead>
<tbody>
<tr>
<td>( h^0 )</td>
<td>( y_i )</td>
<td>( y_i )</td>
</tr>
<tr>
<td>( h^1 )</td>
<td>( (c_1 + c_2)f(y_i, t_i) )</td>
<td>( f(y_i, t_i) )</td>
</tr>
</tbody>
</table>
| \( h^2 \)      | \( c_2 \left[ f_y(y_i, t_i)a_2 f(y_i, t_i) + f_t(y_i, t_i)a_2 \right] \) | \( \left[ f_y(y_i, t_i)f(y_i, t_i) + f_t(y_i, t_i) \right] \frac{1}{2} \)

Thus, we conclude

\[ c_1 + c_2 = 1 \]
\[ c_2a_2 = 1/2 \]

(1.37)

This is a system of two equations in three unknowns or constants \( c_1, c_2, a_2 \); thus, one constant can be selected arbitrarily (there are actually an infinite number of second-order RK methods, depending on the arbitrary choice of one of the constants in Equations 1.37). Here is one choice:

Choose \( c_2 = 1/2 \)
Other constants \( c_1 = 1/2 \) \hspace{1cm} (1.38)
\( a_2 = 1 \)

and the resulting second-order RK method is

\[ y_{i+1} = y_i + c_1k_1 + c_2k_2 = y_i + \frac{k_1 + k_2}{2} \]
\[ k_1 = f(y_i, t_i)h \]
\[ k_2 = f(y_i + a_2k_1(y_i, t_i), t_i + a_2h)h = f(y_i + f(y_i, t_i)h, t_i + h)h \]

which is the modified Euler method, Equations 1.27, 1.28, and 1.29.

For the choice

Choose \( c_2 = 1 \)
Other constants \( c_1 = 0 \) \hspace{1cm} (1.39)
\( a_2 = 1/2 \)

the resulting second-order RK method is

\[ y_{i+1} = y_i + c_1k_1 + c_2k_2 = y_i + k_2 \]
\[ k_1 = f(y_i, t_i)h \]
\[ k_2 = f(y_i + a_2k_1(y_i, t_i), t_i + a_2h)h \]
\[ = f(y_i + (1/2)f(y_i, t_i)h, t_i + (1/2)h)h \]
\[ = f(y_i + (1/2)k_1, t_i + (1/2)h)h \] \hspace{1cm} (1.40c)
which is the midpoint method illustrated in Figure 1.3. As the name suggests, an Euler step is used to compute a predicted value of the solution at the midpoint between points $i$ and $i + 1$ according to Equation 1.40c. The corresponding midpoint derivative ($k_2$ of Equation 1.40c) is then used to advance the solution from $i$ to $i + 1$ (according to Equation 1.40a).

Another choice of the constants in Equation 1.37 is (Iserles, 2 p. 84)

\[
\begin{align*}
\text{Choose} & \quad c_2 = \frac{3}{4} \\
\text{Other constants} & \quad c_1 = \frac{1}{4} \\
& \quad a_2 = \frac{2}{3}
\end{align*}
\]  

(1.41)

and therefore

\[
\begin{align*}
y_{i+1} & = y_i + \frac{1}{4}k_1 + \frac{3}{4}k_2 \\
k_1 & = f(y_i, t_i)h \\
k_2 & = f(y_i + \frac{2}{3}k_1, t_i + \frac{2}{3}h)h
\end{align*}
\]  

(1.42a, 1.42b, 1.42c)

The third-order RK formulas are derived in the same way, but the partial differentiation is more complicated. Thus, we just state the beginning
equations and the final result (Iserles, p. 40). The third order stepping formula is

\[ y_{i+1} = y_i + c_1 k_1 + c_2 k_2 + c_3 k_3 \]  

(1.43a)

The RK constants are

\[ k_1 = f(y_i, t_i)h \]  

(1.43b)

\[ k_2 = f(y_i + a_2 k_1, t_i + a_2 h)h \]  

(1.43c)

\[ k_3 = f(y_i + b_3 k_1 + (a_3 - b_3) k_2, t_i + a_3 h)h \]  

(1.43d)

Four algebraic equations define the six constants \(c_1, c_2, c_3, a_2, a_3, b_3\) (obtained by matching the stepping formula, Equation 1.43a, with the Taylor series up to and including the term \((d^3 y_i/ dt^3)(h^3/3!)\))

\[ c_1 + c_2 + c_3 = 1 \]  

(1.43e)

\[ c_2 a_2 + c_3 a_3 = 1/2 \]  

(1.43f)

\[ c_2 a_2^2 + c_3 a_3^2 = 1/3 \]  

(1.43g)

\[ c_3 (a_3 - b_3) a_2 = 1/6 \]  

(1.43h)

To illustrate the use of Equations 1.43e to 1.43h, we can take \(c_2 = c_3 = \frac{3}{8}\), and from Equation 1.43e, \(c_1 = 1 - \frac{3}{8} - \frac{3}{8} = \frac{2}{8}\). From Equation 1.43f

\[ (3/8)a_2 + (3/8)a_3 = 1/2 \]

or \(a_2 = \frac{4}{3} - a_3\). From Equation 1.43g,

\[ (3/8)(4/3 - a_3)^2 + (3/8)a_3^2 = 1/3 \]

or \(a_3 = \frac{2}{3}\) (by the quadratic formula). Thus, \(a_2 = \frac{4}{3} - \frac{2}{3} = \frac{2}{3}\), and from Equation 1.43h,

\[ (3/8)(2/3 - b_3)2/3 = 1/6 \]

or \(b_3 = 0\).

This particular third-order Nyström method (Iserles, p. 40) is therefore

\[ y_{i+1} = y_i + (2/8) k_1 + (3/8) k_2 + (3/8) k_3 \]  

(1.44a)

\[ k_1 = f(y_i, t_i)h \]  

(1.44b)

\[ k_2 = f(y_i + (2/3) k_1, t_i + (2/3) h)h \]  

(1.44c)

\[ k_3 = f(y_i + (2/3) k_2, t_i + (2/3) h)h \]  

(1.44d)

We next consider some MATLAB code which implements the Euler method of Equation 1.28, the modified Euler method of Equations 1.30, the second-order RK of Equations 1.42, and the third-order RK of Equations 1.44.
The objective is to investigate the accuracy of these RK methods in computing solutions to an ODE test problem.

1.4 Accuracy of RK Methods

We start with the numerical solution of a single ODE, Equation 1.3, subject to initial condition Equation 1.4, by the Euler and modified Euler methods, Equation 1.28 and Equations 1.30. The analytical solution, Equation 1.5, can be used to calculate the exact errors in the numerical solutions.

Equation 1.3 models the growth of tumors, and this important application is first described in the words of Braun1 (the dependent variable in Equation 1.3 is changed from “y” to “V” corresponding to Braun’s notation where V denotes tumor volume).

It has been observed experimentally that “free living” dividing cells, such as bacteria cells, grow at a rate proportional to the volume of the dividing cells at that moment. Let \( V(t) \) denote the volume of the dividing cells at time \( t \). Then,

\[
\frac{dV}{dt} = \lambda V
\]

(1.45)

for some positive constant \( \lambda \). The solution of Equation 1.45 is

\[
V(t) = V_0 e^{\lambda(t-t_0)}
\]

(1.46)

where \( V_0 \) is the volume of dividing cells at the initial time \( t_0 \). Thus, free living dividing cells grow exponentially with time. One important consequence of Equation 1.46 is that the volume of the cells keeps doubling every time interval of length \( \ln 2/\lambda \).

On the other hand, solid tumors do not grow exponentially with time. As the tumor becomes larger, the doubling time of the total tumor volume continuously increases. Various researchers have shown that the data for many solid tumors is fitted remarkably well, over almost a 1000-fold increase in tumor volume, by the equation (previously Equation 1.5)

\[
V(t) = V_0 e^{\lambda(1-exp(-\alpha t))}
\]

(1.47)

where \( \exp(x) = e^x \), and \( \lambda \) and \( \alpha \) are positive constants.

Equation 1.47 is usually referred to as a Gompertzian relation. It says that the tumor grows more and more slowly with the passage of time, and that it ultimately approaches the limiting volume \( V_0 e^{\lambda/\alpha} \). Medical researchers have long been concerned with explaining this deviation from simple exponential growth. A great deal of insight into this problem can be gained by finding a differential equation satisfied by \( V(t) \). Differentiating
Equation 1.47 gives
\[
\frac{dV}{dt} = V_0 \lambda \exp(-\alpha t) \exp\left(\frac{\lambda}{\alpha} (1 - \exp(-\alpha t))\right) \\
= \lambda e^{-\alpha t} V
\] (1.48)
(formerly Equation 1.3).

Two conflicting theories have been advanced for the dynamics of tumor growth. They correspond to the two arrangements
\[
\frac{dV}{dt} = (\lambda e^{-\alpha t}) V \quad (1.48a)
\]
\[
\frac{dV}{dt} = \lambda (e^{-\alpha t}) V \quad (1.48b)
\]

of differential Equation 1.48. According to the first theory, the retarding effect of tumor growth is due to an increase in the mean generation time of the cells, without a change in the proportion of reproducing cells. As time goes on, the reproducing cells mature, or age, and thus divide more slowly. This theory corresponds to the bracketing of Equation 1.48a.

The bracketing of Equation 1.48b suggests the mean generation time of the dividing cells remains constant, and the retardation of growth is due to a loss in reproductive cells in the tumor. One possible explanation for this is that a necrotic region develops in the center of the tumor. This necrosis appears at a critical size for a particular type of tumor, and thereafter, the necrotic “core” increases rapidly as the total tumor mass increases. According to this theory, a necrotic core develops because in many tumors the supply of blood, and thus of oxygen and nutrients, is almost completely confined to the surface of the tumor and a short distance beneath it. As the tumor grows, the supply of oxygen to the central core by diffusion becomes more and more difficult, resulting in the formation of a necrotic core.

We can note the following interesting ideas about this problem:

- Equation 1.48 is a linear, variable coefficient ODE; it can also be considered to have a variable eigenvalue.
- The application of mathematical analysis to tumor dynamics apparently started with a “solution” to an ODE, i.e., Equation 1.47.
- To gain improved insight into tumor dynamics, the question was posed “Is there an ODE corresponding to Equation 1.47?”
- Once an ODE was found (Equation 1.48), it helped explain why the solution, Equation 1.47, represents tumor dynamics so well.
- This is a reversal of the usual process of starting with a differential equation model, then using the solution to explain the performance of the problem system.

A MATLAB program that implements the solution of Equation 1.48 using the Euler and modified Euler methods, Equations 1.28 and 1.30, follows:
% Program 1.1
% Tumor model of eqs. (1.47), (1.48)

% Model parameters
V0=1.0;
lambda=1.0;
alpha=1.0;

% Step through cases
for ncase=1:4

% Integration step
if (ncase==1) h=1.0 ; nsteps=1 ; end
if (ncase==2) h=0.1 ; nsteps=10 ; end
if (ncase==3) h=0.01 ; nsteps=100 ; end
if (ncase==4) h=0.001; nsteps=1000; end

% Variables for ODE integration
tf=10.0;
t=0.0;

% Initial condition
V1=V0;
V2=V0;

% Print heading
fprintf('

h = %6.3f
',h);
fprintf(...
' t Ve V1 errV1 estV1
V2 errV2
')

% Continue integration
while t<0.999*tf

% Take nsteps integration steps
for i=1:nsteps

% Store solution at base point
V1b=V1;
V2b=V2;
tb=t;

% RK constant k1
k11=lambda*exp(-alpha*t)*V1*h;
Program 1.1
MATLAB program for the integration of Equation 1.48 by the modified Euler method of Equations 1.28 and 1.30

We can note the following points about Program 1.1:

• The initial condition and the parameters of Equation 1.48 are first defined (note that % defines a comment in MATLAB):

  ```matlab
  % Model parameters
  V0=1.0;
  lambda=1.0;
  alpha=1.0;
  ```

• The program then steps through four cases corresponding to the integration steps \( h = 1.0, 0.1, 0.01, 0.001: \)

  ```matlab
  % Step through cases
  for ncase=1:4
  ```
% Integration step
if(ncase==1)h=1.0;nsteps=1;end
if(ncase==2)h=0.1;nsteps=10;end
if(ncase==3)h=0.01;nsteps=100;end
if(ncase==4)h=0.001;nsteps=1000;end

For each $h$, the corresponding number of integration steps is $nsteps$. Thus, the product $(h)nsteps = 1$ unit in $t$ for each output from the program; i.e., the output from the program is at $t = 0, 1, 2, \ldots, 10$.

• For each case, the initial and final values of $t$ are defined, i.e., $t = 0, tf = 10$, and the initial condition, $V(0) = V0$ is set to start the solution:

% Variables for ODE integration
tf=10.0;
t=0.0;

% Initial condition
V1=V0;
V2=V0;

Two initial conditions are set, one for the Euler solution, computed as $V1$, and one for the modified Euler solution, $V2$ (subsequently, we will program the solution vector, in this case $[V1 V2]^T$, as a one-dimensional (1D) array).

• A heading indicating the integration step, $h$, and the two numerical solutions is then displayed. “…” indicates a line is to be continued on the next line. (Note: “…” does not work in a character string delineated by single quotes, so the character string in the second fprintf statement has been placed on two lines in order to fit within the available page width; to execute this program, the character string should be returned to one line.)

% Print heading
fprintf('

h = %6.3f
',h);
fprintf(...
' t Ve V1 errV1 estV1
V2 errV2
')

• A while loop then computes the solution until the final time, $tf$, is reached:

% Continue integration
while t<0.999*tf
Of course, at the beginning of the execution, $t = 0$ so the `while` loop continues.

- `nsteps` Euler and modified Euler steps are then taken:

  ```
  % Take nsteps integration steps
  for i=1:nsteps
    % Store solution at base point
    V1b=V1;
    V2b=V2;
    tb=t;
  
  At each point along the solution (point $i$), the solution is stored for subsequent use in the numerical integration.

- The first RK constant, $k_1$, is then computed for each dependent variable in $[V_1 \ V_2]^T$ according to Equation 1.27a:

  ```
  % RK constant k1
  k11=lambda*exp(-alpha*t)*V1*h;
  k12=lambda*exp(-alpha*t)*V2*h;
  ```

  Note that we have used the RHS of the ODE, Equation 1.48, in computing $k_1$. $k_{11}$ is $k_1$ for $V_1$, and $k_{12}$ is $k_1$ for $V_2$. Subsequently, the RK constants will be programmed as 1D arrays, e.g., $[k_1(1) \ k_1(2)]^T$.

- The solution is then advanced from the base point according to Equation 1.28:

  ```
  % RK step
  V1=V1b+k11;
  V2=V2b+k12;
  t=tb+h;
  k22=lambda*exp(-alpha*t)*V2*h;
  ```

  The second RK constant, $k_2$ for $V_2$, is then computed according to Equation 1.27b. At the same time, the independent variable, $t$, is advanced.

- The modified Euler solution, $V_2$, is then computed according to Equation 1.29:

  ```
  % RK step
  V2=V2b+(k12+k22)/2.0;
  t=tb+h;
  ```

  The advance of the independent variable, $t$, was done previously and is therefore redundant; it is done again just to emphasize the advance in $t$.
for the modified Euler method. The end statement ends the loop of nsteps steps, starting with

```
for i=1:nsteps
```

• The exact solution, \( V_e \), is computed from Equation 1.47. The exact error in the Euler solution, \( errV_1 \), and in the modified Euler solution, \( errV_2 \), are then computed. Finally, the difference in the two solutions, \( estV_1 = V_2 - V_1 \), is computed as an estimate of the error in \( V_1 \). The independent variable, \( t \), the two dependent variables, \( V_1, V_2 \), and the three errors, \( errV_1, errV_2, estV_1 \), are then displayed.

```
    %
    % Print solutions and errors
    Ve=V0*exp((lambda/alpha)*(1.0-exp(-alpha*t)));
    errV1=V1-Ve;
    errV2=V2-Ve;
    estV1=V2-V1;
    fprintf('%5.1f%9.4f%9.4f%15.10f%15.10f%9.4f%15.10f
',...
            t,Ve,V1,errV1,estV1,V2,errV2);
```

The output from the `fprintf` statement is considered subsequently.

• The while loop is then terminated, followed by the end of the for loop that sets ncase:

```
    %
    % Continue integration
    end
    %
    % Next case
    end
```

We now consider the output from this program listed below (reformatted slightly to fit on a printed page):

```
h = 1.000

Euler method

<table>
<thead>
<tr>
<th>t</th>
<th>Ve</th>
<th>V1</th>
<th>errV1</th>
<th>estV1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>2.0000</td>
<td>0.1184036125</td>
<td>-0.1321205588</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.7358</td>
<td>0.3615489626</td>
<td>-0.3514091013</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>3.1060</td>
<td>0.5197432882</td>
<td>-0.4929227741</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>3.2606</td>
<td>0.5916944683</td>
<td>-0.5573912375</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>3.3204</td>
<td>0.6203353910</td>
<td>-0.5830821148</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>3.3427</td>
<td>0.6311833526</td>
<td>-0.5928173392</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>3.3510</td>
<td>0.6352171850</td>
<td>-0.5964380611</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>3.3541</td>
<td>0.6367070277</td>
<td>-0.5977754184</td>
</tr>
</tbody>
</table>
```
<table>
<thead>
<tr>
<th>t</th>
<th>Ve</th>
<th>V2</th>
<th>errV2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8679</td>
<td>-0.0137169464</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3843</td>
<td>0.0101398613</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.6131</td>
<td>0.0268205142</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.7033</td>
<td>0.0343032307</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>2.7373</td>
<td>0.0372532762</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>2.7499</td>
<td>0.0383660134</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>2.7546</td>
<td>0.0387791239</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>2.7563</td>
<td>0.0389877746</td>
</tr>
<tr>
<td>9.0</td>
<td>2.7179</td>
<td>2.7569</td>
<td>0.0390084461</td>
</tr>
<tr>
<td>10.0</td>
<td>2.7182</td>
<td>2.7572</td>
<td>0.0390084461</td>
</tr>
</tbody>
</table>

**modified Euler method**

<table>
<thead>
<tr>
<th>t</th>
<th>Ve</th>
<th>V2</th>
<th>errV2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8816</td>
<td>-0.0000409693</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3745</td>
<td>0.0003303677</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.5868</td>
<td>0.0005383591</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.6696</td>
<td>0.0006250422</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>2.7007</td>
<td>0.0006583152</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>2.7122</td>
<td>0.0006707429</td>
</tr>
<tr>
<td>t</td>
<td>Ve</td>
<td>V1</td>
<td>errV1</td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>--------</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>2.7165</td>
<td>0.0006753402</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>2.7180</td>
<td>0.0006770348</td>
</tr>
<tr>
<td>9.0</td>
<td>2.7179</td>
<td>2.7186</td>
<td>0.0006776587</td>
</tr>
<tr>
<td>10.0</td>
<td>2.7182</td>
<td>2.7188</td>
<td>0.0006778883</td>
</tr>
</tbody>
</table>

**Euler method**

<table>
<thead>
<tr>
<th>t</th>
<th>Ve</th>
<th>V1</th>
<th>errV1</th>
<th>estV1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8835</td>
<td>0.0018699629</td>
<td>-0.0018697473</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3786</td>
<td>0.0044269942</td>
<td>-0.0044231149</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.5921</td>
<td>0.0058291952</td>
<td>-0.0058231620</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.6754</td>
<td>0.0064227494</td>
<td>-0.0064158254</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>2.7067</td>
<td>0.0066525021</td>
<td>-0.0066452372</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>2.7183</td>
<td>0.0067386119</td>
<td>-0.0067312197</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>2.7226</td>
<td>0.0067705073</td>
<td>-0.0067630680</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>2.7242</td>
<td>0.0067922704</td>
<td>-0.0067748139</td>
</tr>
<tr>
<td>9.0</td>
<td>2.7179</td>
<td>2.7247</td>
<td>0.0067866019</td>
<td>-0.0067791389</td>
</tr>
<tr>
<td>10.0</td>
<td>2.7182</td>
<td>2.7249</td>
<td>0.0067881959</td>
<td>-0.0067807306</td>
</tr>
</tbody>
</table>

**modified Euler method**

<table>
<thead>
<tr>
<th>t</th>
<th>Ve</th>
<th>V2</th>
<th>errV2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8816</td>
<td>-0.0000000647</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3742</td>
<td>0.0000038793</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.5863</td>
<td>0.0000060332</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.6690</td>
<td>0.0000069239</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>2.7000</td>
<td>0.0000072649</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>2.7116</td>
<td>0.0000073922</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>2.7158</td>
<td>0.0000074392</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>2.7174</td>
<td>0.0000074566</td>
</tr>
<tr>
<td>9.0</td>
<td>2.7179</td>
<td>2.7180</td>
<td>0.0000074629</td>
</tr>
<tr>
<td>10.0</td>
<td>2.7182</td>
<td>2.7182</td>
<td>0.0000074653</td>
</tr>
</tbody>
</table>

**Euler method**

<table>
<thead>
<tr>
<th>t</th>
<th>Ve</th>
<th>V1</th>
<th>errV1</th>
<th>estV1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8818</td>
<td>0.0001878608</td>
<td>-0.0001878611</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3747</td>
<td>0.0004436596</td>
<td>-0.0004436202</td>
</tr>
</tbody>
</table>
modified Euler method

<table>
<thead>
<tr>
<th>t</th>
<th>V1</th>
<th>V2</th>
<th>errV1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8816</td>
<td>-0.0000000003</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3742</td>
<td>0.0000000394</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.5863</td>
<td>0.0000000610</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.6689</td>
<td>0.0000000700</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>2.7000</td>
<td>0.0000000734</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>2.7116</td>
<td>0.0000000747</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>2.7158</td>
<td>0.0000000751</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>2.7174</td>
<td>0.0000000753</td>
</tr>
<tr>
<td>9.0</td>
<td>2.7179</td>
<td>2.7179</td>
<td>0.0000000754</td>
</tr>
<tr>
<td>10.0</td>
<td>2.7182</td>
<td>2.7182</td>
<td>0.0000000754</td>
</tr>
</tbody>
</table>

We can note the following points about this output:

- Considering first the output for the Euler method at $t = 1$:

  - The exact error, $errV1$, decreases linearly with integration step, $h$. For example, when $h$ is decreased from 0.01 to 0.001, $errV1$ decreases from 0.0018696826 to 0.0001878608. Roughly speaking, as the decimal point in $h$ moves one place, the decimal point in $errV1$ moves one place. However, this is true only when $h$ becomes small (so that higher-order terms in the underlying Taylor series become negligibly small).
  - Thus, the error in the Euler method is proportional to $h$

    $$errV1 = Ch^1$$
where $C$ is a constant. The Euler method is therefore termed first order in $h$ or first order correct or of order $h$, which is usually designated as

$$\text{err} V_1 = O(h)$$

where “O” denotes “of order.”

— The estimated error, $\text{est} V_1$ is also first order in $h$ (note again, that as $h$ is decreased by a factor of 1/10, $\text{est} V_1$ decreases by a factor of 1/10). Furthermore, the estimated error, $\text{est} V_1$, approaches the exact error, $\text{err} V_1$ for small $h$. This is an important point since the estimated error can be computed without knowing the exact solution; in other words, we can estimate the error in the numerical solution without knowing the exact solution. The estimated error, $\text{est} V_1 = V_2 - V_1$ is the same as $\varepsilon_1$ given by Equation 1.26b and discussed in words following Equations 1.26.

— If the estimated error, $\text{est} V_1$, is added as a correction to the numerical solution, $V_1$, the corrected solution (in the last column) is much closer to the exact solution, $V_e$. Thus, the estimated error can not only be used to judge the accuracy of the numerical solution, and thereby used to decrease $h$ if necessary to meet a specified error tolerance (see again Equation 1.26b and the subsequent discussion), but the estimated error can be used as a correction for the numerical solution to obtain a more accurate solution. We will make use of these important features of the estimated error in the subsequent routines that automatically adjust the step, $h$, to achieve a specified accuracy.

• Considering next the output for the modified Euler method at $t = 1$:

<table>
<thead>
<tr>
<th>$h$</th>
<th>$V_e$</th>
<th>$V_2$</th>
<th>$\text{err} V_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.8816</td>
<td>1.8679</td>
<td>-0.0137169464</td>
</tr>
<tr>
<td>0.1</td>
<td>1.8816</td>
<td>1.8816</td>
<td>-0.0000409693</td>
</tr>
<tr>
<td>0.01</td>
<td>1.8816</td>
<td>1.8816</td>
<td>-0.0000000647</td>
</tr>
<tr>
<td>0.001</td>
<td>1.8816</td>
<td>1.8816</td>
<td>-0.0000000003</td>
</tr>
</tbody>
</table>

We can note the following points for this output:

— The exact error for the modified Euler method, $\text{err} V_2$, is substantially smaller than the error for the Euler method, $\text{err} V_1$ and $\text{est} V_1$. This is to be expected since the modified Euler method includes the second derivative term in the Taylor series, $(d^2 y/dt^2)(h^2/2!)$, while the Euler method includes only the first derivative term, $(dy/dt)(h/1!)$.

— In other words, the exact error, $\text{err} V_2$, decreases much faster with $h$ than does $\text{err} V_1$. The order of this decrease is difficult to assess from the solution at $t = 1$. For example, when $h$ is decreased from 0.1 to 0.01, the number of zeros after the decimal point increases from four ($-0.000040969$) to seven ($-0.0000000647$) (or roughly, a decrease of $1/1000$). But when $h$ decreases from 0.01 to 0.001, the number of zeros
after the decimal point only increases from seven (−0.0000000647) to nine (−0.0000000003) (or roughly, a decrease of 1/100). Thus, is the order of the modified Euler method $O(h^2)$ or $O(h^3)$?

- We come to a somewhat different conclusion if we consider the modified Euler solution at $t = 10$:

<table>
<thead>
<tr>
<th>$h$</th>
<th>$Ve$</th>
<th>$V2$</th>
<th>$errV2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.7182</td>
<td>2.7572</td>
<td>0.0390084461</td>
</tr>
<tr>
<td>0.1</td>
<td>2.7182</td>
<td>2.7188</td>
<td>0.0006778883</td>
</tr>
<tr>
<td>0.01</td>
<td>2.7182</td>
<td>2.7182</td>
<td>0.0000074653</td>
</tr>
<tr>
<td>0.001</td>
<td>2.7182</td>
<td>2.7182</td>
<td>0.0000000754</td>
</tr>
</tbody>
</table>

We can note the following points for this output:

- The error, $errV2$, now appears to be second order. For example, when $h$ is reduced from 0.1 to 0.01, the error decreases from 0.0006778883 to 0.0000074653, a decrease of approximately 1/100. Similarly, when $h$ is reduced from 0.01 to 0.001, the error decreases from 0.0000074653 to 0.0000000754, again a decrease of approximately 1/100. Thus, we can conclude that at least for this numerical output at $t = 10$, the modified Euler method appears to be second order correct, i.e.,

$$errV2 = O(h^2)$$

We shall generally find this to be the case (the modified Euler method is second order), although, clearly, there can be exceptions (i.e., the output at $t = 1$).

- Finally, we can come to some additional conclusions when comparing the output for the Euler and modified Euler methods:

  - Generally, for both methods, the accuracy of the numerical solutions can be improved by decreasing $h$. This process is termed $h$ refinement, and is an important procedure in ODE library integration routines, i.e., decreasing $h$ to improve the solution accuracy.

  - An error in the numerical solution, in this case $estV1$, can be estimated by subtracting the solutions from two methods of different orders, i.e., $estV1 = V2 − V1$. This estimated error can then be used to adjust $h$ to achieve a solution of prescribed accuracy (see Equations 1.26). This procedure of subtracting solutions of different order is termed $p$ refinement since generally the order of the approximations is stated in terms of a variable “p”, i.e.,

$$error = O(h^p)$$

In the present case, $p = 1$ for the Euler method (it is first order correct), and $p = 2$ for the modified Euler method (it is second order correct). Thus, by using the $p$ refinement of increasing $p$ from 1 to 2,
we can estimate the error in the numerical solution (without having to know the exact solution), and thereby make some adjustments in $h$ to achieve a specified accuracy.

— The integration errors we have been considering are called truncation errors since they result from truncation of the underlying Taylor series (after $(dy/dt)(h/1!)$ and $(d^2y/dt^2)(h^2/2!)$ for the Euler and modified Euler methods, respectively).

— The preceding analysis and conclusions are based on a sufficiently small value of $h$ that the higher-order terms (in $h$) in the Taylor series (after the point of truncation) are negligibly small.

— We have not produced a rigorous proof of $O(h)$ and $O(h^2)$ for the Euler method and modified Euler method. Rather, all of the preceding analysis was through the use of a single, linear ODE, Equation 1.48. Thus, we cannot conclude that these order conditions are generally true (for any system of ODEs). Fortunately, they have been observed to be approximately correct for many ODE systems, both linear and nonlinear.

— Higher-order RK algorithms that fit more of the terms of the underlying Taylor series are available (consider the third-order RK method of Equations 1.44). The preceding error analysis can be applied to them in the same way, and we will now consider again the results for the numerical solution of Equation 1.48. In other words, we can consider $h$ and $p$ refinement for higher-order RK methods.

— The higher order of the modified Euler method, $O(h^2)$, relative to the Euler method, $O(h)$, was achieved through additional computation. Specifically, in the preceding MATLAB program, the Euler method required only one derivative evaluation (use of Equation 1.48) for each step along the solution, while the modified Euler method required two derivative evaluations for each step along the solution. In other words, we pay a “computational price” of additional derivative evaluations when using higher-order methods (that fit more of the underlying Taylor series). However, this additional computation is usually well worth doing (consider the substantially more accurate solution of Equation 1.48 when using the modified Euler method relative to the Euler method, and how much more quickly the error dropped off with decreasing $h$, i.e., $O(h^2)$ vs. $O(h)$ ). Generally, an increase in the order of the method of one (e.g., $O(h)$ to $O(h^2)$) requires one additional derivative evaluation for order up to and including four; beyond fourth order, increasing the order of accuracy by one will require more than one additional derivative evaluation (we shall observe this for a fifth-order RK method to be discussed subsequently).

— In all of the preceding discussion, we have assumed that the solution to an ODE system can be represented by a Taylor series (or a truncated
Taylor series), which is basically a polynomial in $h$. Of course, this
does not have to be the case, but we are assuming that in using
numerical ODE integration algorithms, for sufficiently small $h$, the
Taylor series approximation of the solution is sufficiently accurate
for the given ODE application.

— The RK method is particularly attractive since it can be formulated
for increasing orders (more terms in the Taylor series) without having
to differentiate the differential equation to produce the higher-order
derivatives required in the Taylor series. Thus, all we have to do
in the programming of an ODE system is numerically evaluate the
derivatives defined by the ODEs.

— As we shall see in subsequent examples, the RK method can be ap-
plied to the $n \times n$ problem ($n$ ODEs in $n$ unknowns) as easily as we
applied it to the $1 \times 1$ problem of Equation 1.48. Thus, it is a general
procedure for the solution of systems of ODEs of virtually any order
($n \times n$) and complexity (which is why it is so widely used). In other
words, the RK algorithms (as well as other well-established integra-
tion algorithms) are a powerful tool in the use of ODEs in science
and engineering; we shall see that the same is also true for PDEs.

We now conclude this section by considering the errors in the numerical
solution of Equation 1.48 with a (2, 3) RK pair (i.e., $O(h^2)$ and $O(h^3)$) in analogy
with the (1, 2) pair of the Euler and modified Euler methods), and then a (4, 5)
pair ($O(h^4)$ and $O(h^5)$). This error analysis will establish that the expected
order conditions are realized and also will provide two higher RK pairs that
we can then put into library ODE integration routines.

The (2, 3) pair we considered previously (Equations 1.42 and 1.44) is coded
in the following program. Here we have switched back from the dependent
variable $V$ used previously in Equation 1.48 to the more commonly used $y$ in
Equation 1.3. Also, $y_2$ is the solution of Equation 1.3 using the second-order
RK of Equations 1.42 while $y_3$ is the solution using the third-order RK of
Equations 1.44.

```
% Program 1.2
% Tumor model of eqs. (1.47), (1.48)
% (or eqs. (1.3), (1.4), (1.5))
%
% Model parameters
y0=1.0;
lambda=1.0;
alpha=1.0;
%
% Step through cases
for ncase=1:4
```

Copyright © 2004 by Chapman & Hall/CRC
% Integration step
if(ncase==1)h=1.0 ;nsteps=1 ;end
if(ncase==2)h=0.1 ;nsteps=10 ;end
if(ncase==3)h=0.01 ;nsteps=100 ;end
if(ncase==4)h=0.001;nsteps=1000;end

% Variables for ODE integration
 tf=10.0;
 t=0.0;

% Initial condition
 y2=y0;
 y3=y0;

% Print heading
 fprintf('

h = %6.3f
',h);
 fprintf(...
' t  ye  y2  erry2  esty2
       y3  erry3\n');

% Continue integration
 while t<0.999*tf

% Take nsteps integration steps
 for i=1:nsteps

% Store solution at base point
 y2b=y2;
 y3b=y3;
 tb=t;

% RK constant k1
 k12=lambda*exp(-alpha*t)*y2*h;
 k13=lambda*exp(-alpha*t)*y3*h;

% RK constant k2
 y2=y2b+(2.0/3.0)*k12;
 y3=y3b+(2.0/3.0)*k13;
 t=tb +(2.0/3.0)*h;
 k22=lambda*exp(-alpha*t)*y2*h;
 k23=lambda*exp(-alpha*t)*y3*h;

% RK integration K3
 y3=y3b+(2.0/3.0)*k23;
\[
\begin{align*}
    t &= t_b + (2.0/3.0) \cdot h; \\
    k_{33} &= \lambda \cdot \exp(-\alpha \cdot t) \cdot y_3 \cdot h; \\
    \% \\
    \% \text{RK step} \\
    y_2 &= y_{2b} + (1.0/4.0) \cdot k_{12} + (3.0/4.0) \cdot k_{22}; \\
    y_3 &= y_{3b} + (1.0/4.0) \cdot k_{13} + (3.0/8.0) \cdot k_{23} + (3.0/8.0) \cdot k_{33}; \\
    t &= t_b + h; \\
    \end{align*}
\]

\textbf{Program 1.2}

Program for the integration of Equation 1.48 by the RK (2, 3) pair of Equations 1.42 and 1.44

Program 1.2 closely parallels Program 1.1. The only essential difference is the coding of the RK (2, 3) pair of Equations 1.42 and 1.44 in place of the RK (1, 2) pair of Equations 1.28 and 1.29. We can note the following points about Program 1.2:

- Initial condition (Equation 1.4) is again set for \( y_2 \) and \( y_3 \) to start the numerical solutions:
  \[
  \begin{align*}
  \% \\
  \% \text{Initial condition} \\
  y_2 &= y_0; \\
  y_3 &= y_0; \\
  \end{align*}
  \]
- The integration proceeds with the outer \textit{while} loop (that eventually reaches the final time, \( t_f \)), and an inner \textit{for} loop that takes \textit{nsteps} RK steps for each output. For each pass through the inner loop, the solution is stored at the base point for subsequent use in the RK formulas:
  \[
  \begin{align*}
  \% \\
  \% \text{Continue integration} \\
  \text{while } t < 0.999 \cdot t_f
  \end{align*}
  \]
% Take nsteps integration steps
    for i=1:nsteps
      % Store solution at base point
      y2b=y2;
      y3b=y3;
      tb=t;

      • The RK constant $k_1$ is computed for each dependent variable by using Equation 1.3 ($k_{12}$ for the $k_1$ of $y_2$ and $k_{13}$ for the $k_1$ of $y_3$):
        % RK constant k1
        k12=lambda*exp(-alpha*t)*y2*h;
        k13=lambda*exp(-alpha*t)*y3*h;

      • The solution is then advanced from the base point using a $\frac{2}{3}$ weighting applied to $k_1$ and $h$ (in accordance with Equations 1.42 and 1.44):
        % RK constant k2
        y2=y2b+(2.0/3.0)*k12;
        y3=y3b+(2.0/3.0)*k13;
        t=tb+(2.0/3.0)*h;
        k22=lambda*exp(-alpha*t)*y2*h;
        k23=lambda*exp(-alpha*t)*y3*h;

        This advance of the dependent and independent variables sets the stage for the calculation of $k_2$ (again, using Equation 1.3).

      • $k_3$ is computed for $y_3$ (it is not required for $y_2$):
        % RK integration K3
        y3=y3b+(2.0/3.0)*k23;
        t=tb+(2.0/3.0)*h;
        k33=lambda*exp(-alpha*t)*y3*h;

      • All the required RK constants have now been computed, and the solutions can be advanced to the next point using the stepping formulas:
        % RK step
        y2=y2b+(1.0/4.0)*k12+(3.0/4.0)*k22;
        y3=y3b+(1.0/4.0)*k13+(3.0/8.0)*k23+(3.0/8.0)*k33;
        t=tb+h;
    end

    Note that the stepping formula for $y_2$ does not include $k_3$. The end statement concludes the for loop that is executed nsteps times.
• The solutions, $y_2$ and $y_3$, and associated errors are then displayed:

```matlab
% Print solutions and errors
ye=y0*exp((lambda/alpha)*(1.0-exp(-alpha*t)));
erry2=y2-ye;
erry3=y3-ye;
esty2=y3-y2;
fprintf('%5.1f%9.4f%9.4f%15.10f%15.10f%9.4f
%15.10f
',
t,ye,y2,erry2,esty2,y3,erry3);
```

• Finally, the `while` loop is concluded, followed by the `for` loop that sets the values of $h$, and the initial and final values of $t$:

```matlab
% Continue integration
end
% Next case
end
```

• Note that Equation 1.3 was used twice to compute $k_1$ and $k_2$ for $y_2$ (two derivative evaluations), and Equation 1.3 was used three times to compute $k_1$, $k_2$, and $k_3$ for $y_3$ (three derivative evaluations). This again illustrates the additional computation required, in this case, the calculation of $k_3$, to achieve higher-order results ($O(h^3)$ rather than $O(h^2)$). This improved accuracy is evident in the following output from Program 1.2.

The output from Program 1.2 is listed below (again, with some minor formatting to fit on a printed page):

```matlab
h = 1.000

Second order RK

t    ye    y2    erry2    esty2
1.0  1.8816  1.8918  0.0101750113  -0.0185221389
2.0  2.3742  2.3995  0.0252529307  -0.0352289432
3.0  2.5863  2.6170  0.0307095187  -0.0408693758
4.0  2.6689  2.7014  0.03324302424  -0.0425724266
5.0  2.7000  2.7330  0.0330043494  -0.0431262652
6.0  2.7116  2.7448  0.0332064307  -0.0433188969
7.0  2.7158  2.7491  0.0332794779  -0.0433881911
8.0  2.7174  2.7507  0.0333061722  -0.0434134670
9.0  2.7179  2.7513  0.0333159682  -0.0434227360
10.0 2.7182  2.7515  0.0333195687  -0.0434261419
```
### Third order RK

<table>
<thead>
<tr>
<th>t</th>
<th>ye</th>
<th>y3</th>
<th>erry3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8732</td>
<td>-0.0083471276</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3642</td>
<td>-0.0099760125</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.5761</td>
<td>-0.0101598572</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.6588</td>
<td>-0.0101421842</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>2.6899</td>
<td>-0.0101219158</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>2.7014</td>
<td>-0.0101124662</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>2.7057</td>
<td>-0.0101087132</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>2.7073</td>
<td>-0.0101072948</td>
</tr>
<tr>
<td>9.0</td>
<td>2.7179</td>
<td>2.7078</td>
<td>-0.0101067678</td>
</tr>
<tr>
<td>10.0</td>
<td>2.7182</td>
<td>2.7081</td>
<td>-0.0101065733</td>
</tr>
</tbody>
</table>

### h = 0.100

### Second order RK

<table>
<thead>
<tr>
<th>t</th>
<th>ye</th>
<th>y2</th>
<th>erry2</th>
<th>esty2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8819</td>
<td>0.0003179977</td>
<td>-0.0003270335</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3748</td>
<td>0.0005660244</td>
<td>-0.0005762943</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.5869</td>
<td>0.0006477190</td>
<td>-0.0006581264</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.6696</td>
<td>0.0006733478</td>
<td>-0.0006837363</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>2.7007</td>
<td>0.0006819708</td>
<td>-0.0006923405</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>2.7122</td>
<td>0.0006850226</td>
<td>-0.0006953838</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>2.7165</td>
<td>0.0006861284</td>
<td>-0.0006964862</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>2.7181</td>
<td>0.0006865329</td>
<td>-0.0006968894</td>
</tr>
<tr>
<td>9.0</td>
<td>2.7179</td>
<td>2.7186</td>
<td>0.0006866814</td>
<td>-0.0006970374</td>
</tr>
<tr>
<td>10.0</td>
<td>2.7182</td>
<td>2.7188</td>
<td>0.0006867360</td>
<td>-0.0006970918</td>
</tr>
</tbody>
</table>

### Third order RK

<table>
<thead>
<tr>
<th>t</th>
<th>ye</th>
<th>y3</th>
<th>erry3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8816</td>
<td>-0.0000090358</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3742</td>
<td>-0.0000102699</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.5862</td>
<td>-0.0000104074</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.6689</td>
<td>-0.0000103885</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>2.7000</td>
<td>-0.0000103698</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>2.7115</td>
<td>-0.0000103611</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>2.7158</td>
<td>-0.0000103577</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>2.7174</td>
<td>-0.0000103564</td>
</tr>
<tr>
<td>9.0</td>
<td>2.7179</td>
<td>2.7179</td>
<td>-0.0000103560</td>
</tr>
<tr>
<td>10.0</td>
<td>2.7182</td>
<td>2.7181</td>
<td>-0.0000103558</td>
</tr>
</tbody>
</table>
h = 0.010

Second Order RK

<table>
<thead>
<tr>
<th>t</th>
<th>ye</th>
<th>y2</th>
<th>erry2</th>
<th>esty2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8816</td>
<td>0.00000035779</td>
<td>-0.00000035865</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3742</td>
<td>0.00000062016</td>
<td>-0.00000062112</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.5863</td>
<td>0.00000070634</td>
<td>-0.00000070731</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.6690</td>
<td>0.00000073355</td>
<td>-0.00000073451</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>2.7000</td>
<td>0.00000074275</td>
<td>-0.00000074371</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>2.7116</td>
<td>0.00000074601</td>
<td>-0.00000074697</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>2.7158</td>
<td>0.00000074720</td>
<td>-0.00000074815</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>2.7174</td>
<td>0.00000074763</td>
<td>-0.00000074859</td>
</tr>
<tr>
<td>9.0</td>
<td>2.7179</td>
<td>2.7180</td>
<td>0.00000074779</td>
<td>-0.00000074875</td>
</tr>
<tr>
<td>10.0</td>
<td>2.7182</td>
<td>2.7182</td>
<td>0.00000074785</td>
<td>-0.00000074880</td>
</tr>
</tbody>
</table>

Third order RK

<table>
<thead>
<tr>
<th>t</th>
<th>ye</th>
<th>y3</th>
<th>erry3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8816</td>
<td>-0.0000000085</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3742</td>
<td>-0.0000000096</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.5863</td>
<td>-0.0000000096</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.6689</td>
<td>-0.0000000096</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>2.7000</td>
<td>-0.0000000096</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>2.7116</td>
<td>-0.0000000096</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>2.7158</td>
<td>-0.0000000095</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>2.7174</td>
<td>-0.0000000095</td>
</tr>
<tr>
<td>9.0</td>
<td>2.7179</td>
<td>2.7179</td>
<td>-0.0000000095</td>
</tr>
<tr>
<td>10.0</td>
<td>2.7182</td>
<td>2.7182</td>
<td>-0.0000000095</td>
</tr>
</tbody>
</table>

h = 0.001

Second order RK

<table>
<thead>
<tr>
<th>t</th>
<th>ye</th>
<th>y2</th>
<th>erry2</th>
<th>esty2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8816</td>
<td>0.00000000362</td>
<td>-0.00000000362</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3742</td>
<td>0.00000000626</td>
<td>-0.00000000626</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.5863</td>
<td>0.00000000713</td>
<td>-0.00000000713</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.6689</td>
<td>0.00000000740</td>
<td>-0.00000000740</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>2.7000</td>
<td>0.00000000749</td>
<td>-0.00000000749</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>2.7116</td>
<td>0.00000000752</td>
<td>-0.00000000753</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>2.7158</td>
<td>0.00000000754</td>
<td>-0.00000000754</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>2.7174</td>
<td>0.00000000754</td>
<td>-0.00000000754</td>
</tr>
</tbody>
</table>
Third order RK

\[
\begin{array}{cccc}
 t & ye & y^3 & erry^3 \\
 1.0 & 1.8816 & 1.8816 & 0.0000000000 \\
 2.0 & 2.3742 & 2.3742 & 0.0000000000 \\
 3.0 & 2.5863 & 2.5863 & 0.0000000000 \\
 4.0 & 2.6689 & 2.6689 & 0.0000000000 \\
 5.0 & 2.7000 & 2.7000 & 0.0000000000 \\
 6.0 & 2.7116 & 2.7116 & 0.0000000000 \\
 7.0 & 2.7158 & 2.7158 & 0.0000000000 \\
 8.0 & 2.7174 & 2.7174 & 0.0000000000 \\
 9.0 & 2.7179 & 2.7179 & 0.0000000000 \\
10.0 & 2.7182 & 2.7182 & 0.0000000000 \\
\end{array}
\]

This output closely parallels the previous output for the (1, 2) RK pair. Here are some details.

- Considering the output for the second-order RK at \( t = 1 \):

<table>
<thead>
<tr>
<th>( h )</th>
<th>( ye )</th>
<th>( y^2 )</th>
<th>erry2</th>
<th>esty2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.8816</td>
<td>1.8918</td>
<td>0.0101750113</td>
<td>-0.0185221389</td>
</tr>
<tr>
<td>0.1</td>
<td>1.8816</td>
<td>1.8819</td>
<td>0.0003179977</td>
<td>-0.0003270335</td>
</tr>
<tr>
<td>0.01</td>
<td>1.8816</td>
<td>1.8816</td>
<td>0.0000357799</td>
<td>-0.000035865</td>
</tr>
<tr>
<td>0.001</td>
<td>1.8816</td>
<td>1.8816</td>
<td>0.0000003620</td>
<td>-0.0000003620</td>
</tr>
</tbody>
</table>

- The \( O(h^2) \) behavior of erry2 is clear, i.e., for \( h = 0.1, 0.01, 0.001 \) the corresponding values of erry2 are

\[
0.0003179977, \quad 0.0000357799, \quad 0.0000003620
\]

so that for each reduction in \( h \) by 1/10, erry2 is reduced by a factor of 1/100 (two more zeros are added after the decimal point).

- The same is true for the estimated error, esty2 (computed as the difference \( y^3 - y^2 \)), i.e., for \( h = 0.1, 0.01, 0.001 \) the corresponding values of esty2 are

\[
-0.0003270335, \quad -0.000035865, \quad -0.0000003620
\]

so that two more zeros are added after the decimal point for each 1/10 reduction in \( h \).
The estimated error, \( \text{esty}_2 \), is in close agreement with the exact error, \( \text{erry}_2 \), for small \( h \).

Thus, adding \( \text{esty}_2 \) as a correction to \( y_2 \) will bring the corrected \( y_2 \) into closer agreement with the exact solution, \( y_e \). In other words, \( \text{esty}_2 \) can be used to determine whether \( h \) is small enough to achieve a prescribed accuracy, and once an acceptable \( h \) is thereby selected, \( \text{esty}_2 \) can be added to \( y_2 \) to improve the numerical solution (all without knowledge of the exact solution).

The corresponding output for the third order RK at \( t = 1 \) is

<table>
<thead>
<tr>
<th>( h )</th>
<th>( y_e )</th>
<th>( y^3 )</th>
<th>( \text{erry}^3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.8816</td>
<td>1.8732</td>
<td>-0.0083471276</td>
</tr>
<tr>
<td>0.1</td>
<td>1.8816</td>
<td>1.8816</td>
<td>-0.0000090358</td>
</tr>
<tr>
<td>0.01</td>
<td>1.8816</td>
<td>1.8816</td>
<td>-0.0000000085</td>
</tr>
<tr>
<td>0.001</td>
<td>1.8816</td>
<td>1.8816</td>
<td>0.0000000000</td>
</tr>
</tbody>
</table>

Again, the third order behavior is clear. For \( h = 1, 0.1, 0.01, 0.001 \), the corresponding exact errors are

\(-0.0083471276, -0.0000090358, -0.0000000085, 0.0000000000\)

so a 1/10 reduction in \( h \) results in a 1/1000 reduction in \( \text{erry}^3 \).

In fact, since for most scientific and engineering applications of ODEs, five figure accuracy of the numerical solutions is usually adequate, the last two values of \( \text{erry}^3 \) (for \( h = 0.01, 0.001 \)) can be considered excessively small (these errors are much less than five significant figures compared to the exact solution \( y_e = 1.8816 \)). In other words, \( h = 0.01, 0.001 \) are excessively small. This is an important point. While MATLAB produced all of the numerical output (for \( h = 1, 0.1, 0.01, 0.001 \)) in the order of a second or two for this modest 1x1 problem, for large systems of ODEs, using an excessively small \( h \) will merely result in long computer run times with no significant improvement in the accuracy of the solution. Thus, library routines for integrating ODEs increase \( h \) as well as decrease it to produce solutions close to the specified error tolerance (and not far below the specified error because of excessively small \( h \)). We shall subsequently consider this feature of reducing and increasing \( h \) to stay close to the specified error tolerance in the library routines.

Stated in another way, the preceding solutions for \( h = 1, 0.1, 0.01, 0.001 \) for the interval \( 0 \leq t \leq tf (= 10) \) required 10/1, 10/0.1, 10/0.01, 10/0.001 steps, respectively. 10/0.1 = 100 steps were adequate (because of the accuracy of the third-order RK), while 10/0.01 = 1000
and 10/0.001 = 10000 steps produced excessive accuracy. However, 10/1 = 10 steps were inadequate as might be expected.

- In conclusion, the effectiveness of higher order algorithms, e.g., the third-order RK, in reducing the error in the numerical solution of ODEs is clearly evident from this example.

To conclude this section, we consider a widely used RK (4, 5) pair, the Runge Kutta Fehlberg (RKF) method (Iserles, 2 p. 84):

\[
\begin{align*}
  k_1 &= f(y_i, t_i)h \\
  k_2 &= f(y_i + k_1/4, t_i + h/4)h \\
  k_3 &= f(y_i + (3/32)k_1 + (9/32)k_2, t_i + (3/8)h)h \\
  k_4 &= f(y_i + (1932/2197)k_1 - (7200/2197)k_2 + (7296/2197)k_3, \\
    & \quad t_i + (12/13)h)h \\
  k_5 &= f(y_i + (439/216)k_1 - 8k_2 + (3680/513)k_3 - (845/4104)k_4, \\
    & \quad t_i + h)h \\
  k_6 &= f(y_i - (8/27)k_1 + 2k_2 - (3544/2565)k_3 + (1859/4104)k_4 \\
    & \quad - (11/40)k_5, t_i + (1/2)h)h
\end{align*}
\]

A \(O(h^4)\) stepping formula is then

\[
y_{i+1} = y_i + (25/216)k_1 + (1408/2565)k_3 + (2197/4104)k_4 - (1/5)k_5
\]

and a \(O(h^5)\) stepping formula is (with the same \(k\) terms)

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

An error estimate can then be obtained by subtracting Equation 1.49g from Equation 1.49h:

\[
\epsilon_i = y_{i+1,5} - y_{i+1,4}
\]

Note that six derivative evaluations are required (\(k_1\) through \(k_6\)), even though the final result from Equation 1.49h is only \(O(h^5)\) (the number of derivative evaluations will, in general, be equal to or greater than the order of the final stepping formula).

The stepping formulas of Equations 1.49h and 1.49g match the Taylor series up to and including the terms \((d^4 y_i/dt^4)(h^4/4!)\) and \((d^5 y_i/dt^5)(h^5/5!)\), respectively, as demonstrated by the following Program 1.3.
Program 1.3
% Tumor model of eqs. (1.47), (1.48)
% (or eqs. (1.3), (1.4), (1.5))
% Model parameters
V0=1.0;
lambda=1.0;
alpha=1.0;
% Step through cases
for ncase=1:4

% Integration step
if(ncase==1)h=1.0 ;nsteps=1 ;end
if(ncase==2)h=0.1 ;nsteps=10 ;end
if(ncase==3)h=0.01 ;nsteps=100 ;end
if(ncase==4)h=0.001;nsteps=1000;end

% Variables for ODE integration
tf=10.0;
t=0.0;

% Initial condition
V4=V0;
V5=V0;

% Print heading
fprintf('

h = %6.3f
',h);
fprintf('t Ve V4 errV4 estV4
V5 errV5\n');

% Continue integration
while t<0.999*tf

% Take nsteps integration steps
for i=1:nsteps

% Store solution at base point
V4b=V4;
V5b=V5;
tb=t;

Copyright 2004 by Chapman & Hall/CRC
% RK constant k1
k14=lambda*exp(-alpha*t)*V4*h;
k15=lambda*exp(-alpha*t)*V5*h;
%
% RK constant k2
V4=V4b+0.25*k14;
V5=V5b+0.25*k15;
t= tb+0.25*h;
k24=lambda*exp(-alpha*t)*V4*h;
k25=lambda*exp(-alpha*t)*V5*h;
%
% RK constant k3
V4=V4b+(3.0/32.0)*k14...
  +(9.0/32.0)*k24;
V5=V5b+(3.0/32.0)*k15...
  +(9.0/32.0)*k25;
t= tb+(3.0/8.0)*h;
k34=lambda*exp(-alpha*t)*V4*h;
k35=lambda*exp(-alpha*t)*V5*h;
%
% RK constant k4
V4=V4b+(1932.0/2197.0)*k14...
  -(7200.0/2197.0)*k24...
  +(7296.0/2197.0)*k34;
V5=V5b+(1932.0/2197.0)*k15...
  -(7200.0/2197.0)*k25...
  +(7296.0/2197.0)*k35;
t= tb+(12.0/13.0)*h;
k44=lambda*exp(-alpha*t)*V4*h;
k45=lambda*exp(-alpha*t)*V5*h;
%
% RK constant k5
V4=V4b+(439.0/216.0)*k14...
  -(8.0    )*k24...
  +(3680.0/513.0)*k34...
  -(845.0/5104.0)*k44;
V5=V5b+(439.0/216.0)*k15...
  -(8.0    )*k25...
  +(3680.0/513.0)*k35...
  -(845.0/5104.0)*k45;
t= tb+h;
k54=lambda*exp(-alpha*t)*V4*h;
k55=lambda*exp(-alpha*t)*V5*h;
%
% RK constant k6
\[ V_4 = V_{4b} - \left( \frac{8.0}{27.0} \right) k_{14} + \left( \frac{2.0}{1} \right) k_{24} - \left( \frac{3544.0}{2565.0} \right) k_{34} + \left( \frac{1859.0}{4104.0} \right) k_{44} - \left( \frac{11.0}{40.0} \right) k_{54}; \]

\[ V_5 = V_{5b} - \left( \frac{8.0}{27.0} \right) k_{15} + \left( \frac{2.0}{1} \right) k_{25} - \left( \frac{3544.0}{2565.0} \right) k_{35} + \left( \frac{1859.0}{4104.0} \right) k_{45} - \left( \frac{11.0}{40.0} \right) k_{55}; \]

\[ T = T_b + 0.5h; \]

\[ k_{65} = \lambda \exp(-\alpha T) \cdot V_5 \cdot h; \]

\[ V_4 = V_{4b} + \left( \frac{25.0}{216.0} \right) k_{14} + \left( \frac{1408.0}{2565.0} \right) k_{34} + \left( \frac{2197.0}{4104.0} \right) k_{44} - \left( \frac{1.0}{5.0} \right) k_{54}; \]

\[ V_5 = V_{5b} + \left( \frac{16.0}{135.0} \right) k_{15} + \left( \frac{6656.0}{12825.0} \right) k_{35} + \left( \frac{28561.0}{56430.0} \right) k_{45} - \left( \frac{9.0}{50.0} \right) k_{55} + \left( \frac{2.0}{55.0} \right) k_{65}; \]

\[ T = T_b + h; \]

% RK step

% Print solutions and errors
Ve = V0 * exp((\lambda / \alpha) * (1.0 - exp(-\alpha T)));
errV4 = V4 - Ve;
errV5 = V5 - Ve;
estV4 = V5 - V4;
fprintf('%5.1f%9.4f%9.4f%15.10f%15.10f%9.4f%15.10f\n', ...
t, Ve, V4, errV4, estV4, V5, errV5);

% Continue integration

% Next case

Program 1.3

Program for the integration of Equation 1.48 by the RKF45 method of Equations 1.49

Program 1.3 closely parallels Programs 1.1 and 1.2. Therefore, we consider only the essential difference, the evaluation of the RK constants, \( k_1 \) to \( k_6 \):
% Store solution at base point
y4b=y4;
y5b=y5;
tb=t;

% RK constant k1
k14=lambda*exp(-alpha*t)*y4*h;
k15=lambda*exp(-alpha*t)*y5*h;

% RK constant k2
k24=lambda*exp(-alpha*t)*y4*h;
k25=lambda*exp(-alpha*t)*y5*h;

% RK constant k3
k34=lambda*exp(-alpha*t)*y4*h;
k35=lambda*exp(-alpha*t)*y5*h;

% RK constant k4
k44=lambda*exp(-alpha*t)*y4*h;
k45=lambda*exp(-alpha*t)*y5*h;

% RK constant k5
k54=lambda*exp(-alpha*t)*y4*h;
k55=lambda*exp(-alpha*t)*y5*h;
\[-(\frac{845.0}{4104.0})k45;\]
\[t = tb + h;\]
\[k4 = \lambda \exp(-\alpha t)y4h;\]
\[k5 = \lambda \exp(-\alpha t)y5h;\]
\[
\% \%
\%
RK constant k6
\]
\[y4 = y4b - (\frac{8.0}{27.0})k14 - (\frac{2.0}{27.0})k24 + (\frac{2.0}{27.0})k34 - (\frac{3544.0}{2565.0})k44 + (\frac{1859.0}{4104.0})k54;\]
\[y5 = y5b - (\frac{8.0}{27.0})k15 + (\frac{2.0}{27.0})k25 - (\frac{3544.0}{2565.0})k35 + (\frac{1859.0}{4104.0})k45 - (\frac{11.0}{40.0})k55;\]
\[t = tb + 0.5h;\]
\[k6 = \lambda \exp(-\alpha t)y5h;\]
\[
\% \%
\%
RK step
\]
\[y4 = y4b + (\frac{25.0}{216.0})k14 + (\frac{1408.0}{2565.0})k34 + (\frac{2197.0}{4104.0})k44 - (\frac{1.0}{5.0})k54;\]
\[y5 = y5b + (\frac{16.0}{135.0})k15 + (\frac{6656.0}{12825.0})k35 + (\frac{28561.0}{56430.0})k45 - (\frac{9.0}{50.0})k55 + (\frac{2.0}{55.0})k65;\]
\[t = tb + h;\]
\end

Not much explanation is required for this code since it follows directly from Equations 1.49a to 1.49i. We can note the following points:

- Clearly there is a substantial degree of repetitive coding that could be streamlined through the use of 1D arrays (particularly in the calculation of $k_1$ to $k_6$).
- The $O(h^4)$ and $O(h^5)$ solutions are computed independently, and we will next observe that they can be combined.
- The code is a mixture of problem-specific coding, i.e., using Equations 1.3 and 1.4, and general coding, i.e., Equations 1.49a to 1.49i. The separation of the code into problem-specific and general coding would facilitate the application of the $(4, 5)$ pair to other problems; we will see how this can be done; i.e., we are headed toward the development of general library routines.

Copyright 2004 by Chapman & Hall/CRC
The output from Program 1.3 is listed below (again, reformatted slightly to fit on a printed page):

\[ h = 1.000 \]

**Fourth order method**

<table>
<thead>
<tr>
<th>t</th>
<th>ye</th>
<th>y4</th>
<th>erry4</th>
<th>esty4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8814</td>
<td>-0.0001703991</td>
<td>0.0000660329</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3740</td>
<td>-0.0002465843</td>
<td>0.0001027923</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.5860</td>
<td>-0.0002711657</td>
<td>0.0001138050</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.6687</td>
<td>-0.0002799191</td>
<td>0.0001183974</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>2.6997</td>
<td>-0.0002831506</td>
<td>0.0001202510</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>2.7113</td>
<td>-0.0002843446</td>
<td>0.0001209619</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>2.7155</td>
<td>-0.0002847848</td>
<td>0.0001212276</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>2.7171</td>
<td>-0.0002849469</td>
<td>0.0001213259</td>
</tr>
<tr>
<td>9.0</td>
<td>2.7179</td>
<td>2.7178</td>
<td>-0.0002850065</td>
<td>0.0001213621</td>
</tr>
<tr>
<td>10.0</td>
<td>2.7182</td>
<td>2.7179</td>
<td>-0.0002850285</td>
<td>0.0001213755</td>
</tr>
</tbody>
</table>

**Fifth order method**

<table>
<thead>
<tr>
<th>t</th>
<th>ye</th>
<th>y5</th>
<th>erry5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8815</td>
<td>-0.0001043662</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3741</td>
<td>-0.0001437920</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.5861</td>
<td>-0.0001573607</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.6688</td>
<td>-0.0001615234</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>2.6999</td>
<td>-0.0001628996</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>2.7114</td>
<td>-0.0001633827</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>2.7156</td>
<td>-0.0001635572</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>2.7172</td>
<td>-0.0001636210</td>
</tr>
<tr>
<td>9.0</td>
<td>2.7179</td>
<td>2.7178</td>
<td>-0.0001636444</td>
</tr>
<tr>
<td>10.0</td>
<td>2.7182</td>
<td>2.7180</td>
<td>-0.0001636530</td>
</tr>
</tbody>
</table>

\[ h = 0.100 \]

**Fourth order method**

<table>
<thead>
<tr>
<th>t</th>
<th>ye</th>
<th>y4</th>
<th>erry4</th>
<th>esty4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8816</td>
<td>-0.0000000138</td>
<td>0.0000000135</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3742</td>
<td>-0.0000000198</td>
<td>0.0000000192</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.5863</td>
<td>-0.0000000218</td>
<td>0.0000000212</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.6689</td>
<td>-0.0000000226</td>
<td>0.0000000220</td>
</tr>
</tbody>
</table>

Copyright 2004 by Chapman & Hall/CRC
<table>
<thead>
<tr>
<th>t</th>
<th>ye</th>
<th>y5</th>
<th>erry5</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>2.7000</td>
<td>-0.0000000229</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>2.7116</td>
<td>-0.0000000230</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>2.7158</td>
<td>-0.0000000231</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>2.7174</td>
<td>-0.0000000231</td>
</tr>
<tr>
<td>9.0</td>
<td>2.7179</td>
<td>2.7179</td>
<td>-0.0000000231</td>
</tr>
<tr>
<td>10.0</td>
<td>2.7182</td>
<td>2.7182</td>
<td>-0.0000000231</td>
</tr>
</tbody>
</table>

Fifth order method

<table>
<thead>
<tr>
<th>t</th>
<th>ye</th>
<th>y5</th>
<th>erry5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8816</td>
<td>-0.0000000003</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3742</td>
<td>-0.0000000005</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.5863</td>
<td>-0.0000000006</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.6689</td>
<td>-0.0000000006</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>2.7000</td>
<td>-0.0000000006</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>2.7116</td>
<td>-0.0000000006</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>2.7158</td>
<td>-0.0000000006</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>2.7174</td>
<td>-0.0000000006</td>
</tr>
<tr>
<td>9.0</td>
<td>2.7179</td>
<td>2.7179</td>
<td>-0.0000000006</td>
</tr>
<tr>
<td>10.0</td>
<td>2.7182</td>
<td>2.7182</td>
<td>-0.0000000006</td>
</tr>
</tbody>
</table>

h = 0.010

Fourth order method

<table>
<thead>
<tr>
<th>t</th>
<th>ye</th>
<th>y4</th>
<th>erry4</th>
<th>esty4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8816</td>
<td>0.00000000000</td>
<td>0.00000000000</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3742</td>
<td>0.00000000000</td>
<td>0.00000000000</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.5863</td>
<td>0.00000000000</td>
<td>0.00000000000</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.6689</td>
<td>0.00000000000</td>
<td>0.00000000000</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>2.7000</td>
<td>0.00000000000</td>
<td>0.00000000000</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>2.7116</td>
<td>0.00000000000</td>
<td>0.00000000000</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>2.7158</td>
<td>0.00000000000</td>
<td>0.00000000000</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>2.7174</td>
<td>0.00000000000</td>
<td>0.00000000000</td>
</tr>
<tr>
<td>9.0</td>
<td>2.7179</td>
<td>2.7179</td>
<td>0.00000000000</td>
<td>0.00000000000</td>
</tr>
<tr>
<td>10.0</td>
<td>2.7182</td>
<td>2.7182</td>
<td>0.00000000000</td>
<td>0.00000000000</td>
</tr>
</tbody>
</table>

Fifth order method

<table>
<thead>
<tr>
<th>t</th>
<th>ye</th>
<th>y5</th>
<th>erry5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8816</td>
<td>0.00000000000</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3742</td>
<td>0.00000000000</td>
</tr>
</tbody>
</table>
h = 0.001

### Fourth order method

<table>
<thead>
<tr>
<th>t</th>
<th>ye</th>
<th>y4</th>
<th>erry4</th>
<th>esty4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8816</td>
<td>0.0000000000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3742</td>
<td>0.0000000000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.5863</td>
<td>0.0000000000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.6689</td>
<td>0.0000000000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>2.7000</td>
<td>0.0000000000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>2.7116</td>
<td>0.0000000000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>2.7158</td>
<td>0.0000000000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>2.7174</td>
<td>0.0000000000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>9.0</td>
<td>2.7179</td>
<td>2.7179</td>
<td>0.0000000000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>10.0</td>
<td>2.7182</td>
<td>2.7182</td>
<td>0.0000000000</td>
<td>0.0000000000</td>
</tr>
</tbody>
</table>

### Fifth order method

<table>
<thead>
<tr>
<th>t</th>
<th>ye</th>
<th>y5</th>
<th>erry5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8816</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3742</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.5863</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.6689</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>2.7000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>2.7116</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>2.7158</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>2.7174</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>9.0</td>
<td>2.7179</td>
<td>2.7179</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>10.0</td>
<td>2.7182</td>
<td>2.7182</td>
<td>0.0000000000</td>
</tr>
</tbody>
</table>

This output is relatively easy to discuss since there are a lot of zeros! Specifically,
• At \( t = 1 \), with \( h = 1 \), which corresponds to a total of \( 10/1 = 10 \) steps, the \( O(h^3) \) method computed a solution accurate to at least four figures!

\[
\begin{array}{cccc}
1.0 & 1.8816 & 1.8815 & -0.0001043662 \\
\end{array}
\]

With 100 steps \( (h = 0.1) \) the error is only \(-0.0000000003\) at \( t = 1 \)

\[
\begin{array}{cccc}
1.0 & 1.8816 & 1.8816 & -0.0000000003 \\
\end{array}
\]

• The \( O(h^4) \) behavior of the fourth order method is evident (at least to a degree). At \( t = 1 \),

\[
\begin{array}{cccc}
h & ye & y4 & erry4 & esty4 \\
1 & 1.8816 & 1.8814 & -0.0001703991 & 0.0000660329 \\
0.1 & 1.8816 & 1.8816 & -0.0000000138 & 0.0000000135 \\
0.01 & 1.8816 & 1.8816 & 0.0000000000 & 0.0000000000 \\
0.001 & 1.8816 & 1.8816 & 0.0000000000 & 0.0000000000 \\
\end{array}
\]

Approximately four zeros are added to the exact and estimated errors when \( h \) is reduced from 1 to 0.1. When four more zeros (between \( h = 0.1 \) and \( h = 0.01 \)) are added, the error drops below 0.0000000000 corresponding to the \%15.10f format of the \texttt{fprintf} statement in Program 1.3. Clearly, we can conclude that \( h = 0.01, 0.001 \) are excessively small for most practical applications in science and engineering.

• The \( O(h^5) \) behavior of the fifth order method is evident (also, to a degree). At \( t = 1 \),

\[
\begin{array}{cccc}
h & ye & y5 & erry5 \\
1 & 1.8816 & 1.8815 & -0.0001043662 \\
0.1 & 1.8816 & 1.8816 & -0.0000000003 \\
0.01 & 1.8816 & 1.8816 & 0.0000000000 \\
0.001 & 1.8816 & 1.8816 & 0.0000000000 \\
\end{array}
\]

At least five zeros are added to the exact error when \( h \) is reduced from 1 to 0.1. When five more zeros (between \( h = 0.1 \) and \( h = 0.01 \)) are added, the error drops far below 0.0000000000 (presumably) corresponding to the \%15.10f format of the \texttt{fprintf} statement in Program 1.3. Again, we can conclude that \( h = 0.01, 0.001 \) are excessively small for most practical applications, and a library routine would be far more efficient if it limited the reduction in \( h \) to somewhere in the range \( 0.1 \leq h \leq 1 \) rather than allow \( h \) to drop much below 0.1.

• We can conclude the additional effort to compute the RK constants \( k_1 \) to \( k_6 \) is probably worthwhile since far larger steps \( (h) \) can be used to achieve
a solution of a given accuracy than when using lower-order methods (the (1, 2) pair or even the (2, 3) pair).

Finally, we can consider why the various RK algorithms have the orders we have observed (beyond just observing that the higher-order methods fit more of the terms in the underlying Taylor series). For example, why is the first-order RK (Euler’s method) $O(h)$?

The first-order RK method includes the $(dy/dt)(h/1!)$ term in the Taylor series, but excludes through truncation of the Taylor series the term $(d^2y/dt^2) \times (h^2/2!)$ and higher-order terms. If the second order term is considered, the principal source of the integration error for the Euler method, which is true for small $h$ for which the higher-order terms are negligible, then it would seem that the Euler method is second order (because of the $h^2$ in $(d^2y/dt^2)(h^2/2!)$).

However, this second derivative term is the local one step error, that is, the error incurred by taking just one step along the solution of length $h$. But in computing a numerical solution using, for example, Programs 1.1 to 1.3, many steps are taken, and we are primarily interested in the total or global error after many steps (this is the error that we actually observe in the numerical solution to an ODE system, and which we want to control at some acceptable level).

We can analyze the relationship between the local error and the global error in the following way. If we assume that the error in using the Euler method is due to just the second derivative term:

$$
\epsilon_i = \frac{d^2y_i}{dt^2} \frac{h^2}{2!}
$$

then the local or one step error is $O(h^2)$. If we integrate over a series of steps of length $h$ from $t = a$ to $t = b$ using $n$ steps, that is

$$
n = \frac{b - a}{h}
$$

we can then estimate the total or global error as

$$
global \ error = \text{(one step error)(number of steps)}
$$

or

$$
global \ error = \frac{d^2y_i}{dt^2} \frac{h^2}{2!} \left( \frac{b - a}{h} \right) = \frac{d^2y_i}{dt^2} \left( \frac{b - a}{2!} \right) h
$$

so that the global error is $O(h)$ as we observed. Note that this is an approximate analysis based on two assumptions:

1. All of the local error is contained in just the one term $(d^2y_i/dt^2)(h^2/2!)$.
2. The derivative $d^2y_i/dt^2$ is essentially constant over the interval $a \leq t \leq b$ (or we can use some appropriate average value of this second derivative).
A more rigorous analysis to show that the Euler method is $O(h)$ globally is rather involved. Also, generally for the higher-order methods, the global error will be one order in $h$ lower than the one step error, so, for example, the previous $(4, 5)$ pair is $O(h^5) - O(h^6)$ locally, but $O(h^4) - O(h^5)$ globally (again, this can be established in a nonrigorous way for a general interval $t = a$ to $t = b$ as we did for the Euler method).

We now consider the streamlining of the programming as mentioned previously for the $(4, 5)$ pair.

### 1.5 Embedded RK Algorithms

We first note the interesting property of the RKF $(4, 5)$ pair that the RK constants $k_1$ to $k_5$ given by Equations 1.49a to 1.49e are the same for both the $O(h^4)$ and $O(h^5)$ stepping formulas of Equations 1.49g and 1.49h ($k_6$ is required for only the $O(h^5)$ method of Equation 1.49h). Thus, we can consider the $O(h^4)$ method of Equation 1.49g to be embedded in the $O(h^5)$ method of Equation 1.49h. This has an important implication: $k_1$ to $k_5$ need be calculated only once for both methods (rather than for each method as in Program 1.3). With this idea in mind, the only difference between the two methods is the calculation of $k_6$ for the $O(h^5)$ method of Equation 1.49g, and the selection of a base point for the next step (either the $O(h^4)$ or the $O(h^5)$ base point—we will select the latter).

This same feature appears in the $(1, 2)$ pair of Equations 1.28 and 1.29; the Euler method is embedded in the modified Euler method, with the common $k_1$ of Equation 1.27a. Similarly, for the $(2, 3)$ pair, the second-order method of Equation 1.42a is embedded in the third-order method of Equation 1.44a, with the common $k_1$ of Equation 1.42b (or Equation 1.44b) and the common $k_2$ of Equation 1.42c (or Equation 1.44c).

The embedding of the $(1, 2)$ pair is illustrated by the following Program 1.4, which is a small revision of Program 1.1:

```matlab
% Program 1.4
% Tumor model of eqs. (1.47), (1.48)
% (or eqs. (1.3), (1.4), (1.5))
% Model parameters
y0=1.0;
lambda=1.0;
alpha=1.0;

% Step through cases
for ncase=1:4
```

Copyright 2004 by Chapman & Hall/CRC
% Integration step
if(ncase==1)h=1.0 ;nsteps=1 ;end
if(ncase==2)h=0.1 ;nsteps=10 ;end
if(ncase==3)h=0.01 ;nsteps=100 ;end
if(ncase==4)h=0.001;nsteps=1000;end

% Variables for ODE integration
tf=10.0;
t=0.0;

% Initial condition
y2=y0;

% Print heading
fprintf('

h = %6.3f
',h);
fprintf(...
    ' t   ye   y1   erry1   esty1
    y2   erry2\n')

% Continue integration
while t<0.999*tf

% Take nsteps integration steps
for i=1:nsteps

% Store solution at base point
yb=y2;
tb=t;

% RK constant k1
k1=lambda*exp(-alpha*tb)*y2*h;

% RK constant k2
y2=yb+k1;
t=tb+h;
k2=lambda*exp(-alpha*t)*y2*h;

% RK step
y1=yb+k1;
esty1=y2-y1;
y2=yb+(k1+k2)/2.0;
end

% Print solutions and errors
ye=y0*exp((lambda/alpha)*(1.0-exp(-alpha*t)));


Copyright © 2004 by Chapman & Hall/CRC
Program 1.4
Program for the integration of Equation 1.48 by the embedded ((1, 2) pair) modified Euler method of Equations 1.28 and 1.29

We can note the following points about Program 1.4:

- The essential differences between Programs 1.1 and 1.2 are in the way that the RK constants are computed and used. In particular, while keeping in mind that \( y_1 \) is the \( O(h) \) (Euler method) and \( y_2 \) is the \( O(h^2) \) (modified Euler method), the base point is selected as the running value of \( y_2 \):

\[
\%
\% Store solution at base point
yb=y2;
tb=t;
\]

where the initial value of \( y_2 \) was set previously as an initial condition.

- \( k_1 \) and \( k_2 \) are then calculated (according to Equations 1.27a and 1.27b):

\[
\%
\% RK constant \( k_1 \)
k1=lambda*exp(-alpha*tb)*y2*h;
\%
\% RK constant \( k_2 \)
y2=yb+k1;
t=tb+h;
k2=lambda*exp(-alpha*t)*y2*h;
\]

- The first- and second-order stepping formulas are then used (according to Equations 1.28 and 1.29):

\[
\%
\% RK step
y1=yb+k1;
\]
\begin{verbatim}
  y2 = yb + (k1 + k2) / 2.0;
  estyl = y2 - y1;
end
\end{verbatim}

Note in this code that:
- The estimated error in \( y_1 \), \( estyl \), is computed by \( p \) refinement (subtraction of the \( O(h) \) solution from the \( O(h^2) \) solution).
- The same value of \( k_1 \) is used for both the first- and second-order stepping formulas (making use of the embedding of the (2, 3) pair, i.e., the first-order method is embedded in the second-order method)
- The \textit{end} statement terminates the \textit{for} loop of \textit{nsteps} of length \( h \).

Otherwise the programming is essentially the same as in Program 1.1. The output from Program 1.4 is listed below (formatted to fit on a page):

\begin{verbatim}
h = 1.000

First order method
\begin{tabular}{cccccc}
 t & ye & yl & erryl & estyl \\
 1.0 & 1.8816 & 2.0000 & 0.1184036125 & -0.1321205588 \\
 2.0 & 2.3742 & 2.5550 & 0.1808239664 & -0.1706841052 \\
 3.0 & 2.5863 & 2.7070 & 0.1207761366 & -0.0939562225 \\
 4.0 & 2.6689 & 2.7432 & 0.0742305143 & -0.039927836 \\
 5.0 & 2.7000 & 2.7528 & 0.0527351769 & -0.0154819007 \\
 6.0 & 2.7116 & 2.7557 & 0.0441724614 & -0.0058064480 \\
 7.0 & 2.7158 & 2.7567 & 0.0409303986 & -0.0021512746 \\
 8.0 & 2.7174 & 2.7571 & 0.0397250855 & -0.0007934762 \\
 9.0 & 2.7179 & 2.7572 & 0.0392799583 & -0.0002921837 \\
10.0 & 2.7182 & 2.7573 & 0.0391159724 & -0.0001075263 \\
\end{tabular}

Second order method
\begin{tabular}{cccc}
 t & ye & y2 & erry2 \\
 1.0 & 1.8816 & 1.8679 & -0.0137169464 \\
 2.0 & 2.3742 & 2.3843 & 0.0101398613 \\
 3.0 & 2.5863 & 2.6131 & 0.0268205142 \\
 4.0 & 2.6689 & 2.7033 & 0.0341032307 \\
 5.0 & 2.7000 & 2.7373 & 0.0372532762 \\
 6.0 & 2.7116 & 2.7499 & 0.0383660134 \\
 7.0 & 2.7158 & 2.7546 & 0.0388791239 \\
 8.0 & 2.7174 & 2.7563 & 0.0389316092 \\
 9.0 & 2.7179 & 2.7569 & 0.0389877746 \\
10.0 & 2.7182 & 2.7572 & 0.0390084461 \\
\end{tabular}
\end{verbatim}

Copyright 2004 by Chapman & Hall/CRC
**First order method**

<table>
<thead>
<tr>
<th>$t$</th>
<th>$y_e$</th>
<th>$y_1$</th>
<th>$err_1$</th>
<th>$est_y_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8837</td>
<td>0.0021070298</td>
<td>-0.0021479991</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3760</td>
<td>0.0017594212</td>
<td>-0.0014290535</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.5874</td>
<td>0.0011768161</td>
<td>-0.0006384570</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.6698</td>
<td>0.0008767242</td>
<td>-0.0002516819</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>2.7008</td>
<td>0.0007532605</td>
<td>-0.0000949453</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>2.7123</td>
<td>0.0007059944</td>
<td>-0.0000352515</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>2.7165</td>
<td>0.000683524</td>
<td>-0.0000130123</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>2.7181</td>
<td>0.0006818277</td>
<td>-0.0000047929</td>
</tr>
<tr>
<td>9.0</td>
<td>2.7179</td>
<td>2.7186</td>
<td>0.0006794227</td>
<td>-0.0000017640</td>
</tr>
<tr>
<td>10.0</td>
<td>2.7182</td>
<td>2.7188</td>
<td>0.0006785373</td>
<td>-0.0000006491</td>
</tr>
</tbody>
</table>

**Second order method**

<table>
<thead>
<tr>
<th>$t$</th>
<th>$y_e$</th>
<th>$y_2$</th>
<th>$err_2$</th>
<th>$est_y_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8816</td>
<td>-0.0000409693</td>
<td></td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3742</td>
<td>0.0003303677</td>
<td></td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.5863</td>
<td>0.0005383591</td>
<td></td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.6696</td>
<td>0.0006250422</td>
<td></td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>2.7007</td>
<td>0.0006583152</td>
<td></td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>2.7123</td>
<td>0.0006707429</td>
<td></td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>2.7165</td>
<td>0.0006753402</td>
<td></td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>2.7180</td>
<td>0.0006770348</td>
<td></td>
</tr>
<tr>
<td>9.0</td>
<td>2.7179</td>
<td>2.7186</td>
<td>0.0006776587</td>
<td></td>
</tr>
<tr>
<td>10.0</td>
<td>2.7182</td>
<td>2.7188</td>
<td>0.0006778883</td>
<td></td>
</tr>
</tbody>
</table>

**First order method**

<table>
<thead>
<tr>
<th>$t$</th>
<th>$y_e$</th>
<th>$y_1$</th>
<th>$err_1$</th>
<th>$est_y_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8816</td>
<td>0.0000217778</td>
<td>-0.0000218425</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3742</td>
<td>0.0000178106</td>
<td>-0.0000139313</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.5863</td>
<td>0.0000121768</td>
<td>-0.0000061436</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.6690</td>
<td>0.0000093347</td>
<td>-0.0000024108</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>2.7000</td>
<td>0.0000081729</td>
<td>-0.0000009079</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>2.7116</td>
<td>0.0000077291</td>
<td>-0.0000003369</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>2.7158</td>
<td>0.0000075636</td>
<td>-0.0000001243</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>2.7174</td>
<td>0.0000075024</td>
<td>-0.0000000458</td>
</tr>
</tbody>
</table>

Copyright 2004 by Chapman & Hall/CRC


Second order method

<table>
<thead>
<tr>
<th>t</th>
<th>ye</th>
<th>y2</th>
<th>erry2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8816</td>
<td>-0.0000000647</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3742</td>
<td>0.00000038793</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.5863</td>
<td>0.00000060332</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.6690</td>
<td>0.00000069239</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>2.7000</td>
<td>0.00000072649</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>2.7116</td>
<td>0.00000073922</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>2.7158</td>
<td>0.00000074392</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>2.7174</td>
<td>0.00000074566</td>
</tr>
<tr>
<td>9.0</td>
<td>2.7179</td>
<td>2.7179</td>
<td>0.00000074629</td>
</tr>
<tr>
<td>10.0</td>
<td>2.7182</td>
<td>2.7182</td>
<td>0.00000074653</td>
</tr>
</tbody>
</table>

First order method

<table>
<thead>
<tr>
<th>t</th>
<th>ye</th>
<th>y1</th>
<th>erry1</th>
<th>esty1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8816</td>
<td>0.0000002185</td>
<td>-0.0000002187</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3742</td>
<td>0.0000001784</td>
<td>-0.0000001390</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.5863</td>
<td>0.0000001222</td>
<td>-0.0000000612</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.6689</td>
<td>0.0000000940</td>
<td>-0.0000000240</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>2.7000</td>
<td>0.0000000824</td>
<td>-0.0000000090</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>2.7116</td>
<td>0.0000000780</td>
<td>-0.0000000034</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>2.7158</td>
<td>0.0000000764</td>
<td>-0.0000000012</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>2.7174</td>
<td>0.0000000758</td>
<td>-0.0000000005</td>
</tr>
<tr>
<td>9.0</td>
<td>2.7179</td>
<td>2.7179</td>
<td>0.0000000756</td>
<td>-0.0000000002</td>
</tr>
<tr>
<td>10.0</td>
<td>2.7182</td>
<td>2.7182</td>
<td>0.0000000755</td>
<td>-0.0000000001</td>
</tr>
</tbody>
</table>

Second order method

<table>
<thead>
<tr>
<th>t</th>
<th>ye</th>
<th>y2</th>
<th>erry2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8816</td>
<td>-0.0000000003</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3742</td>
<td>0.0000000394</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.5863</td>
<td>0.0000000610</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.6689</td>
<td>0.0000000700</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>2.7000</td>
<td>0.0000000734</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>2.7116</td>
<td>0.0000000747</td>
</tr>
</tbody>
</table>
We can note the following points about this output:

- The first-order method appears to be higher than $O(h)$. For example, at $t = 1$, the output is

<table>
<thead>
<tr>
<th>$h$</th>
<th>$y_e$</th>
<th>$y_1$</th>
<th>erry1</th>
<th>esty1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.8816</td>
<td>2.0000</td>
<td>0.1184036125</td>
<td>-0.1321205588</td>
</tr>
<tr>
<td>0.1</td>
<td>1.8816</td>
<td>1.8837</td>
<td>0.0021070298</td>
<td>-0.0021479991</td>
</tr>
<tr>
<td>0.01</td>
<td>1.8816</td>
<td>1.8816</td>
<td>0.0000217778</td>
<td>-0.0000218425</td>
</tr>
<tr>
<td>0.001</td>
<td>1.8816</td>
<td>1.8816</td>
<td>0.0000002185</td>
<td>-0.0000002187</td>
</tr>
</tbody>
</table>

In fact, the first-order method appears to be second order correct! For example, reducing $h$ from 0.1 to 0.01 reduces the exact error from 0.0021070298 to 0.0000217778 (two zeros are added after the decimal point). The reason for this is that the second-order solution, $y_2$, is used as the base point for the next step along the solution, i.e.,

% Store solution at base point
yb=y2;
tb=t;

To state this in other words, $y_1$ is corrected by $esty1$ before going on to the next point. For example, at $t = 1$ for $h = 0.1$,

$$y_1 + y_{1est} = 1.8837 - 0.0021479991 = 1.8816 = y_2$$

This is an important point discussed previously as Step 4 in the algorithm after Equation 1.26c. In other words, in a library ODE integrator, which automatically adjusts the step $h$, the estimated error $esty1$ will generally be computed to determine if the step $h$ is small enough to satisfy a specified error tolerance. When $h$ becomes small enough to meet the error criterion, the estimated error can be added as a correction before taking the next step along the solution. In this case (the $(1, 2)$ pair), this in effect increases the accuracy of the solution from $O(h)$ to $O(h^2)$ as we observed in the preceding output from Program 1.4.

- This error correction could be programmed in a slightly different, but equivalent, way (see Equations 1.30):
Clearly, adding the estimated error as a correction \((y_2 = y_1 + esty_1)\) before taking the next step along the solution (as explained in the algorithm after Equation 1.26c) was worth doing (the first-order method becomes effectively second order).

- The exact error in \(y_2\) from Program 1.4, \(erry_2\), at \(t = 1\) appears to be greater than \(O(h^2)\):

<table>
<thead>
<tr>
<th>(h)</th>
<th>(ye)</th>
<th>(y_2)</th>
<th>(erry_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.8816</td>
<td>1.8679</td>
<td>−0.0137169464</td>
</tr>
<tr>
<td>0.1</td>
<td>1.8816</td>
<td>1.8816</td>
<td>−0.00000409693</td>
</tr>
<tr>
<td>0.01</td>
<td>1.8816</td>
<td>1.8816</td>
<td>−0.0000000647</td>
</tr>
<tr>
<td>0.001</td>
<td>1.8816</td>
<td>1.8816</td>
<td>−0.0000000003</td>
</tr>
</tbody>
</table>

However, generally, this error is \(O(h^2)\). For example, at \(t = 2\), the exact error in \(y_2\) is

<table>
<thead>
<tr>
<th>(h)</th>
<th>(erry_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.0101398613</td>
</tr>
<tr>
<td>0.1</td>
<td>0.0003303677</td>
</tr>
<tr>
<td>0.01</td>
<td>0.0000038793</td>
</tr>
<tr>
<td>0.001</td>
<td>0.0000000394</td>
</tr>
</tbody>
</table>

(so that two zeros are added after the decimal point for each 1/10 reduction in \(h\), as expected).

- The estimated error, \(esty_1\) appears to converge to the exact error, \(erry_1\), for small \(h\):

<table>
<thead>
<tr>
<th>(h)</th>
<th>(erry1)</th>
<th>(esty1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.1184036125</td>
<td>−0.1321205588</td>
</tr>
<tr>
<td>0.1</td>
<td>0.0021070298</td>
<td>−0.0021479991</td>
</tr>
<tr>
<td>0.01</td>
<td>0.0000217778</td>
<td>−0.0000218425</td>
</tr>
<tr>
<td>0.001</td>
<td>0.0000002185</td>
<td>−0.0000002187</td>
</tr>
</tbody>
</table>

This convergence again illustrates the important point that the estimated error accurately estimates the exact error for small \(h\) (and thus adding it as a correction to the \(O(h)\) solution gives a substantially better solution).
We next investigate the embedding of the (2, 3) pair of Equations 1.42 and 1.44. The following Program 1.5, which is analogous to Program 1.4, illustrates how this embedding can be used.

```matlab
% Program 1.5
% Tumor model of eqs. (1.47), (1.48)
% (or eqs. (1.3), (1.4), (1.5))
%
% Model parameters
y0=1.0;
lambda=1.0;
alpha=1.0;
%
% Step through cases
for ncase=1:4
%
% Integration step
if(ncase==1)h=1.0 ;nsteps=1 ;end
if(ncase==2)h=0.1 ;nsteps=10 ;end
if(ncase==3)h=0.01 ;nsteps=100 ;end
if(ncase==4)h=0.001;nsteps=1000;end
%
% Variables for ODE integration
tf=10.0;
t=0.0;
%
% Initial condition
y3=y0;
%
% Print heading
fprintf('

h = %6.3f
',h);
fprintf(...
' t ye y2 erry2 esty2
y3 erry3\n')
%
% Continue integration
while t<0.999*tf
%
% Take nsteps integration steps
for i=1:nsteps
%
% Store solution at base point
yb=y3;
tb=t;
```
% RK constant k1
k1=lambda*exp(-alpha*t)*y3*h;
%
% RK constant k2
y3=yb+(2.0/3.0)*k1;
t=tb+(2.0/3.0)*h;
k2=lambda*exp(-alpha*t)*y3*h;
%
% RK integration K3
y3=yb+(2.0/3.0)*k2;
t=tb+(2.0/3.0)*h;
k3=lambda*exp(-alpha*t)*y3*h;
%
% RK step
y2=yb+(1.0/4.0)*k1+(3.0/4.0)*k2;
y3=yb+(1.0/4.0)*k1+(3.0/8.0)*k2+(3.0/8.0)*k3;
esty2=y3-y2;
t=tb+h;
end
%
% Print solutions and errors
ye=y0*exp((lambda/alpha)*(1.0-exp(-alpha*t)));
erry2=y2-ye;
erry3=y3-ye;
fprintf('%5.1f%9.4f%9.4f%15.10f%15.10f%9.4f%15.10f
','
t,ye,y2,erry2,esty2,y3,erry3);
%
% Continue integration
end
%
% Next case
end

Program 1.5
Program for the integration of Equation 1.48 by the (2, 3) pair of Equations 1.42 and 1.44

Program 1.5 closely parallels Program 1.4. As expected, $k_1$ and $k_2$ are used for both the second- and third-order stepping formulas ($k_3$ is required for only the third-order stepping formula):
%
% RK step
y2=yb+(1.0/4.0)*k1+(3.0/4.0)*k2;
y3=yb+(1.0/4.0)*k1+(3.0/8.0)*k2+(3.0/8.0)*k3;
esty2=y3-y2;
t=tb+h;
end

The output from Program 1.5 is listed below:

\[ h = 1.000 \]

**Second order method**

<table>
<thead>
<tr>
<th>t</th>
<th>ye</th>
<th>y2</th>
<th>erry2</th>
<th>esty2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8918</td>
<td>0.0101750113</td>
<td>-0.0185221389</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3760</td>
<td>0.0017600371</td>
<td>-0.0117360496</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.5785</td>
<td>-0.0077128645</td>
<td>-0.0024469927</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.6592</td>
<td>-0.0097573439</td>
<td>-0.0003848403</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>2.6900</td>
<td>-0.0100669280</td>
<td>-0.0000549878</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>2.7014</td>
<td>-0.0101048752</td>
<td>-0.0000075909</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>2.7057</td>
<td>-0.0101076784</td>
<td>-0.0000010348</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>2.7073</td>
<td>-0.0101071543</td>
<td>-0.000001404</td>
</tr>
<tr>
<td>9.0</td>
<td>2.7179</td>
<td>2.7078</td>
<td>-0.0101067488</td>
<td>-0.000000190</td>
</tr>
<tr>
<td>10.0</td>
<td>2.7182</td>
<td>2.7081</td>
<td>-0.0101065707</td>
<td>-0.000000026</td>
</tr>
</tbody>
</table>

**Third order method**

<table>
<thead>
<tr>
<th>t</th>
<th>ye</th>
<th>y3</th>
<th>erry3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8732</td>
<td>-0.0083471276</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3642</td>
<td>-0.0099760125</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.5761</td>
<td>-0.0101598572</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.6588</td>
<td>-0.0101421842</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>2.6899</td>
<td>-0.0101219158</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>2.7014</td>
<td>-0.0101124662</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>2.7057</td>
<td>-0.0101087132</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>2.7073</td>
<td>-0.0101072948</td>
</tr>
<tr>
<td>9.0</td>
<td>2.7179</td>
<td>2.7078</td>
<td>-0.0101067678</td>
</tr>
<tr>
<td>10.0</td>
<td>2.7182</td>
<td>2.7081</td>
<td>-0.0101065733</td>
</tr>
</tbody>
</table>

\[ h = 0.100 \]

**Second order method**

<table>
<thead>
<tr>
<th>t</th>
<th>ye</th>
<th>y2</th>
<th>erry2</th>
<th>esty2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8816</td>
<td>0.0000183521</td>
<td>-0.0000273880</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3742</td>
<td>-0.0000035143</td>
<td>-0.0000067556</td>
</tr>
</tbody>
</table>
### Third order method

<table>
<thead>
<tr>
<th>t</th>
<th>ye</th>
<th>y3</th>
<th>erry3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8816</td>
<td>-0.0000090358</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3742</td>
<td>-0.0000102699</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.5862</td>
<td>-0.0000104074</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.6689</td>
<td>-0.0000103885</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>2.7000</td>
<td>-0.0000103698</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>2.7115</td>
<td>-0.0000103611</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>2.7158</td>
<td>-0.0000103577</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>2.7174</td>
<td>-0.0000103564</td>
</tr>
<tr>
<td>9.0</td>
<td>2.7179</td>
<td>2.7179</td>
<td>-0.0000103560</td>
</tr>
<tr>
<td>10.0</td>
<td>2.7182</td>
<td>2.7181</td>
<td>-0.0000103558</td>
</tr>
</tbody>
</table>

\[ h = 0.010 \]

### Second order method

<table>
<thead>
<tr>
<th>t</th>
<th>ye</th>
<th>y2</th>
<th>erry2</th>
<th>esty2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8816</td>
<td>0.0000000183</td>
<td>-0.0000000269</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3742</td>
<td>-0.0000000033</td>
<td>-0.0000000063</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.5863</td>
<td>-0.0000000086</td>
<td>-0.0000000010</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.6689</td>
<td>-0.0000000094</td>
<td>-0.0000000001</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>2.7000</td>
<td>-0.0000000095</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>2.7116</td>
<td>-0.0000000095</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>2.7158</td>
<td>-0.0000000095</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>2.7174</td>
<td>-0.0000000095</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>9.0</td>
<td>2.7179</td>
<td>2.7179</td>
<td>-0.0000000095</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>10.0</td>
<td>2.7182</td>
<td>2.7182</td>
<td>-0.0000000095</td>
<td>0.0000000000</td>
</tr>
</tbody>
</table>
Third order method

<table>
<thead>
<tr>
<th>t</th>
<th>ye</th>
<th>y3</th>
<th>erry3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8816</td>
<td>-0.0000000085</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3742</td>
<td>-0.0000000096</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.5863</td>
<td>-0.0000000096</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.6689</td>
<td>-0.0000000096</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>2.7000</td>
<td>-0.0000000096</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>2.7116</td>
<td>-0.0000000096</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>2.7158</td>
<td>-0.0000000095</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>2.7174</td>
<td>-0.0000000095</td>
</tr>
<tr>
<td>9.0</td>
<td>2.7179</td>
<td>2.7179</td>
<td>-0.0000000095</td>
</tr>
<tr>
<td>10.0</td>
<td>2.7182</td>
<td>2.7182</td>
<td>-0.0000000095</td>
</tr>
</tbody>
</table>

h = 0.001

Second order method

<table>
<thead>
<tr>
<th>t</th>
<th>ye</th>
<th>y2</th>
<th>erry2</th>
<th>esty2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8816</td>
<td>0.0000000000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3742</td>
<td>0.0000000000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.5863</td>
<td>0.0000000000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.6689</td>
<td>0.0000000000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>2.7000</td>
<td>0.0000000000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>2.7116</td>
<td>0.0000000000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>2.7158</td>
<td>0.0000000000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>2.7174</td>
<td>0.0000000000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>9.0</td>
<td>2.7179</td>
<td>2.7179</td>
<td>0.0000000000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>10.0</td>
<td>2.7182</td>
<td>2.7182</td>
<td>0.0000000000</td>
<td>0.0000000000</td>
</tr>
</tbody>
</table>

Third order method

<table>
<thead>
<tr>
<th>t</th>
<th>ye</th>
<th>y3</th>
<th>erry3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8816</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3742</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.5863</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.6689</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>2.7000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>2.7116</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>2.7158</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>2.7174</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>9.0</td>
<td>2.7179</td>
<td>2.7179</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>10.0</td>
<td>2.7182</td>
<td>2.7182</td>
<td>0.0000000000</td>
</tr>
</tbody>
</table>
We can note the following points about this output:

- The corrected $y_2$ appears to be $O(h^3)$, e.g., for $t = 1$,

<table>
<thead>
<tr>
<th>$h$</th>
<th>$y_e$</th>
<th>$y_2$</th>
<th>$erry_2$</th>
<th>$esty_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.8816</td>
<td>1.8918</td>
<td>0.010175113</td>
<td>0.0185221389</td>
</tr>
<tr>
<td>0.1</td>
<td>1.8816</td>
<td>1.8816</td>
<td>0.0000183521</td>
<td>0.0000273880</td>
</tr>
<tr>
<td>0.01</td>
<td>1.8816</td>
<td>1.8816</td>
<td>0.0000000183</td>
<td>0.0000000269</td>
</tr>
<tr>
<td>0.001</td>
<td>1.8816</td>
<td>1.8816</td>
<td>0.0000000000</td>
<td>0.0000000000</td>
</tr>
</tbody>
</table>

Note that when $h$ is reduced from 0.1 to 0.01, $erry_2$ is reduced from 0.0000183521 to 0.0000000183 so that three zeros were added after the decimal point. This is expected since $y_2$ is corrected by $esty_2$ before taking the next step along the solution, thereby giving an $O(h^3)$ result ($y_3$ is used as the base point value for the next step).

- $esty_2$ is not as reliable an estimate of the true error, $erry_2$ as we would like. For some points along the solution, it underestimates the exact error in $y_2$, and at other points, it overestimates the exact error. For example, when $h = 0.01$,

$$h = 0.010$$

Second order method

<table>
<thead>
<tr>
<th>$t$</th>
<th>$y_e$</th>
<th>$y_2$</th>
<th>$erry_2$</th>
<th>$esty_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8816</td>
<td>0.0000000183</td>
<td>0.0000000269</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3742</td>
<td>-0.0000000033</td>
<td>-0.0000000063</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.5863</td>
<td>-0.0000000086</td>
<td>-0.0000000010</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.6689</td>
<td>-0.0000000094</td>
<td>-0.0000000001</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>2.7000</td>
<td>-0.0000000095</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>2.7116</td>
<td>-0.0000000095</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>2.7158</td>
<td>-0.0000000095</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>2.7174</td>
<td>-0.0000000095</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>9.0</td>
<td>2.7179</td>
<td>2.7179</td>
<td>-0.0000000095</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>10.0</td>
<td>2.7182</td>
<td>2.7182</td>
<td>-0.0000000095</td>
<td>0.0000000000</td>
</tr>
</tbody>
</table>

An overestimate of the exact error is conservative in adjusting $h$, but an underestimate will possibly produce an $h$ that is too large to actually limit the exact error to a specified value or tolerance. Certainly we would like to have a reliable (quantitatively correct) estimate of the true error so that we can reliably adjust $h$. Another way to interpret this result (estimate of limited accuracy) is to observe that the higher-order solution $y_3$ also has some error and therefore the estimated error is not a perfect correction (it does not give a higher-order solution without error).
Finally, we investigate the embedding of the (4, 5) pair of Equations 1.49. The following Program 1.6, which is analogous to Programs 1.4 and 1.5, illustrates how this embedding can be used.

```matlab
% Program 1.6
% Tumor model of eqs. (1.47), (1.48)
% (or eqs. (1.3), (1.4), (1.5))
% Model parameters
y0=1.0;
lambda=1.0;
alpha=1.0;
% Step through cases
for ncase=1:4
  % Integration step
  if(ncase==1)h=1.0 ;nsteps=1 ;end
  if(ncase==2)h=0.1 ;nsteps=10 ;end
  if(ncase==3)h=0.01 ;nsteps=100 ;end
  if(ncase==4)h=0.001;nsteps=1000;end
  % Variables for ODE integration
  tf=10.0;
t=0.0;
  % Initial condition
  y5=y0;
  % Print heading
  fprintf('

  h = %6.3f
',h);
  fprintf(...
    '  t  ye  y4  erry4  esty4  y5  erry5\n')
  % Continue integration
  while t<0.999*tf
    % Take nsteps integration steps
    for i=1:nsteps
      % Store solution at base point
      yb=y5;
tb=t;
```
% RK constant k1
k1=lambda*exp(-alpha*t)*y5*h;
%
% RK constant k2
y5=yb+0.25*k1;
t=tb+0.25*h;
k2=lambda*exp(-alpha*t)*y5*h;
%
% RK constant k3
y5=yb+(3.0/32.0)*k1...(9.0/32.0)*k2;
t=tb+(3.0/8.0)*h;
k3=lambda*exp(-alpha*t)*y5*h;
%
% RK constant k4
y5=yb+(1932.0/2197.0)*k1...(7200.0/2197.0)*k2...(7296.0/2197.0)*k3;
t=tb+(12.0/13.0)*h;
k4=lambda*exp(-alpha*t)*y5*h;
%
% RK constant k5
y5=yb+(439.0/216.0)*k1...(-8.0)*k2...+(3680.0/513.0)*k3...(-845.0/4104.0)*k4;
t=tb+h;
k5=lambda*exp(-alpha*t)*y5*h;
%
% RK constant k6
y5=yb-(8.0/27.0)*k1...+(2.0)*k2...-(3544.0/2565.0)*k3...+(1859.0/4104.0)*k4...(-11.0/40.0)*k5;
t=tb+0.5*h;
k6=lambda*exp(-alpha*t)*y5*h;
%
% RK step
y4=yb+(25.0/216.0)*k1...+(1408.0/2565.0)*k3...+(2197.0/4104.0)*k4...(-1.0/5.0)*k5;
\[ y_5 = y_b + \left( \frac{16.0}{135.0} \right) k_1 \]
\[ + \left( \frac{6656.0}{12825.0} \right) k_3 \]
\[ + \left( \frac{28561.0}{56430.0} \right) k_4 \]
\[ - \left( \frac{9.0}{50.0} \right) k_5 \]
\[ + \left( \frac{2.0}{55.0} \right) k_6; \]
\]
\[ \text{esty}_4 = y_5 - y_4; \]
\[ t = t_b + h; \]
\]
\% Printer solutions and errors
\]
\[ y_e = y_0 \exp \left( \frac{\lambda}{\alpha} \left( 1.0 - \exp(-\alpha t) \right) \right); \]
\[ \text{erry}_4 = y_4 - y_e; \]
\[ \text{erry}_5 = y_5 - y_e; \]
\[ \text{fprintf} \left( \%
\[ '\%5.1f\%9.4f\%9.4f\%15.10f\%15.10f\%9.4f\%15.10f\n', ...
\[ t, y_e, y_4, \text{erry}_4, \text{esty}_4, y_5, \text{erry}_5); \]
\]
\% Continue integration
\]
\% Next case
\]
Program 1.6
Program for the integration of Equation 1.48 by the RKF45 pair of Equations 1.49

Program 1.6 closely parallels Programs 1.4 and 1.5. As expected, \( k_1 \) to \( k_5 \) are used for both the fourth- and fifth-order stepping formulas (\( k_6 \) is required for only the fifth-order stepping formula)

\% RK step
\%
\[ y_4 = y_b + \left( \frac{25.0}{216.0} \right) k_1 \]
\[ + \left( \frac{1408.0}{2565.0} \right) k_3 \]
\[ + \left( \frac{2197.0}{4104.0} \right) k_4 \]
\[ - \left( \frac{1.0}{5.0} \right) k_5; \]
\[ y_5 = y_b + \left( \frac{16.0}{135.0} \right) k_1 \]
\[ + \left( \frac{6656.0}{12825.0} \right) k_3 \]
\[ + \left( \frac{28561.0}{56430.0} \right) k_4 \]
\[ - \left( \frac{9.0}{50.0} \right) k_5 \]
\[ + \left( \frac{2.0}{55.0} \right) k_6; \]
\[ \text{esty}_4 = y_5 - y_4; \]
\[ t = t_b + h; \]
\]
Note also that \( k_2 \) is not used in either stepping formula (but it is required to calculate \( k_3 \) to \( k_6 \)).
The output from Program 1.6 is listed below:

\[ h = 1.000 \]

**Fourth order method**

<table>
<thead>
<tr>
<th>t</th>
<th>y4</th>
<th>( \text{err} y4 )</th>
<th>( \text{esty} y4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>-0.0001703991</td>
<td>0.0000660329</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>-0.0001632647</td>
<td>0.0000194727</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>-0.0001591927</td>
<td>0.0000018321</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>-0.0001624755</td>
<td>0.0000009521</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>-0.0001633761</td>
<td>0.0000004765</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>-0.0001635804</td>
<td>0.0000000760</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>-0.0001636494</td>
<td>0.0000000284</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7179</td>
<td>-0.0001636549</td>
<td>0.0000000105</td>
</tr>
<tr>
<td>9.0</td>
<td>2.7182</td>
<td>-0.0001636569</td>
<td>0.0000000039</td>
</tr>
</tbody>
</table>

**Fifth order method**

<table>
<thead>
<tr>
<th>t</th>
<th>y5</th>
<th>( \text{err} y5 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>-0.0001043662</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>-0.0001437920</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>-0.0001573607</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>-0.0001615234</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>-0.0001628996</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>-0.0001633827</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>-0.0001635752</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>-0.0001636210</td>
</tr>
<tr>
<td>9.0</td>
<td>2.7179</td>
<td>-0.0001636444</td>
</tr>
<tr>
<td>10.0</td>
<td>2.7182</td>
<td>-0.0001636530</td>
</tr>
</tbody>
</table>

\[ h = 0.100 \]

**Fourth order method**

<table>
<thead>
<tr>
<th>t</th>
<th>y4</th>
<th>( \text{err} y4 )</th>
<th>( \text{esty} y4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>-0.0000000009</td>
<td>0.0000000006</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>-0.0000000006</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>-0.0000000006</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>-0.0000000006</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>-0.0000000006</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>-0.0000000006</td>
<td>0.0000000000</td>
</tr>
</tbody>
</table>
Fifth order method

<table>
<thead>
<tr>
<th>t</th>
<th>ye</th>
<th>y5</th>
<th>erry5</th>
<th>esty5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8816</td>
<td>-0.0000000003</td>
<td>0.0000000006</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3742</td>
<td>-0.0000000005</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.5863</td>
<td>-0.0000000006</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.6689</td>
<td>-0.0000000006</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>2.7000</td>
<td>-0.0000000006</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>2.7116</td>
<td>-0.0000000006</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>2.7158</td>
<td>-0.0000000006</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>2.7174</td>
<td>-0.0000000006</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>9.0</td>
<td>2.7179</td>
<td>2.7179</td>
<td>-0.0000000006</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>10.0</td>
<td>2.7182</td>
<td>2.7182</td>
<td>-0.0000000006</td>
<td>0.0000000000</td>
</tr>
</tbody>
</table>

h = 0.010

Fourth order method

<table>
<thead>
<tr>
<th>t</th>
<th>ye</th>
<th>y4</th>
<th>erry4</th>
<th>esty4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8816</td>
<td>-0.0000000009</td>
<td>0.0000000006</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3742</td>
<td>-0.0000000006</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.5863</td>
<td>-0.0000000006</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.6689</td>
<td>-0.0000000006</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>2.7000</td>
<td>-0.0000000006</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>2.7116</td>
<td>-0.0000000006</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>2.7158</td>
<td>-0.0000000006</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>2.7174</td>
<td>-0.0000000006</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>9.0</td>
<td>2.7179</td>
<td>2.7179</td>
<td>-0.0000000006</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>10.0</td>
<td>2.7182</td>
<td>2.7182</td>
<td>-0.0000000006</td>
<td>0.0000000000</td>
</tr>
</tbody>
</table>

Fifth order method

<table>
<thead>
<tr>
<th>t</th>
<th>ye</th>
<th>y5</th>
<th>erry5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8816</td>
<td>-0.0000000003</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3742</td>
<td>-0.0000000005</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.5863</td>
<td>-0.0000000006</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.6689</td>
<td>-0.0000000006</td>
</tr>
</tbody>
</table>
h = 0.001

Fourth order method

<table>
<thead>
<tr>
<th>t</th>
<th>ye</th>
<th>y4</th>
<th>erry4</th>
<th>esty4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8816</td>
<td>0.0000000000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3742</td>
<td>0.0000000000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.5863</td>
<td>0.0000000000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.6689</td>
<td>0.0000000000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>2.7000</td>
<td>0.0000000000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>2.7116</td>
<td>0.0000000000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>2.7158</td>
<td>0.0000000000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>2.7174</td>
<td>0.0000000000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>9.0</td>
<td>2.7179</td>
<td>2.7179</td>
<td>0.0000000000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>10.0</td>
<td>2.7182</td>
<td>2.7182</td>
<td>0.0000000000</td>
<td>0.0000000000</td>
</tr>
</tbody>
</table>

Fifth order method

<table>
<thead>
<tr>
<th>t</th>
<th>ye</th>
<th>y5</th>
<th>erry5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8816</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3742</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.5863</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.6689</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>2.7000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>2.7116</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>2.7158</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>2.7174</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>9.0</td>
<td>2.7179</td>
<td>2.7179</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>10.0</td>
<td>2.7182</td>
<td>2.7182</td>
<td>0.0000000000</td>
</tr>
</tbody>
</table>

We can note the following points about this output:

- For \( h = 1 \), esty4 is not a reliable estimate of the exact error, erry4. However, the solution for \( h = 1 \) results from only a total of ten steps in the interval \( 0 \leq t \leq 10 \), so we might expect that the estimated error esty4 will not be very accurate. Also, even with just ten steps, the (4, 5) pair produced a solution that is accurate to about five figures.
• For $h = 0.1$, the solution is so accurate that the exact and estimated errors appear in only the tenth decimal place (the final decimal place provided by the %15.10f format of the fprintf statement).
• $h = 0.01$ and $0.001$ appear to be excessively small.

We conclude this section with a (2, 4) embedded pair, i.e., an $O(h^2)$ method embedded in an $O(h^4)$ method. The fourth-order method (the original RK method reported by Runge and Kutta in the 1890s) is

\begin{align}
  k_1 &= f(y_i, t_i)h \\
  k_2 &= f(y_i + \frac{k_1}{2}, t_i + \frac{h}{2})h \\
  k_3 &= f(y_i + \frac{k_2}{2}, t_i + \frac{h}{2})h \\
  k_4 &= f(y_i + k_3, t_i + h)h
\end{align}

with the stepping formula

\[
y_{i+1} = y_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)
\]

As we discussed previously, Equation 1.50e fits the Taylor series up to and including the fourth-order derivative term, $(d^4y/dt^4)(h^4/4!)$; i.e., the resulting numerical solution is $O(h^4)$.

The second-order midpoint RK method of Equations 1.40 has the same $k_1$ and $k_2$ and therefore is embedded in the fourth-order method. An error estimate for this second-order method can be obtained by subtracting the second-order stepping formula from the fourth-order stepping formula of Equation 1.50e:

\[
  \epsilon_i = y_{4,i+1} - y_{2,i+1} = y_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) - (y_i + k_2) \\
  = (1/6)(k_1 - 4k_2 + 2k_3 + k_4)
\]

Note how the $k_1$ and $k_2$ terms combine in arriving at Equation 1.51 since they are the same for both algorithms, i.e., Equations 1.40 and 1.50. $\epsilon_i$ of Equation 1.51 can now be used to automatically adjust the integration step, $h$, which is the basis of the programming in a set of routines to be discussed in subsequent chapters.

Note also that since this error estimate was achieved by subtracting the stepping formula for a second-order method (Equation 1.40a), from the stepping formula for a fourth-order method (Equation 1.50e), the error estimate actually represents two terms in the Taylor series, i.e., $(d^3y/dt^3)(h^3/6)$ and $(d^4y/dt^4)(h^4/24)$; i.e., $\epsilon_i$ from Equation 1.51 is a two term error estimate, and therefore we might expect that it will be more accurate than the one term error estimates of the preceding (1, 2), (2, 3), and (4, 5) pairs. Experience has indicated this is the case. In fact, some additional embedded RK pairs are listed in Appendix A, which have three term error estimates.
The principal conclusions from this discussion of embedded methods are as follows:

- The RK constants generally can be computed once for both the lower-order and the higher-order methods of an embedded pair. In other words, the common RK constants are the basis for embedded pairs.
- Correction of the lower-order solution using the estimated error (the difference between the higher- and lower-order methods) 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.

1.6 Library ODE Integrators

We have discussed several RK pairs ((1, 2), (2, 3), (2, 4), (4, 5)) that can be used in library routines. Because each pair produces an estimate of the truncation error, these four methods can be used to automatically adjust $h$ to achieve a specified error tolerance. Furthermore, although the preceding programming of the four pairs has been for a 1x1 problem (Equations 1.3 and 1.4 with the analytical solution (Equation 1.5)), they can be applied directly to the $n \times n$ problem by using vectors for the RK constants and the stepping formulas. Thus, we now have everything we need for general-purpose ODE integration routines, which are discussed in subsequent chapters.

To illustrate what we might accomplish, we consider briefly the ODE library routines in MATLAB (of the programming languages considered in the subsequent discussion, only Maple and MATLAB have built-in ODE utilities). MATLAB includes utilities for stiff and nonstiff ODEs (stiffness and stability are discussed briefly in the next section). However, we consider here only the nonstiff MATLAB integrators, ode23 and ode45, which is consistent with the four RK pairs discussed previously since they are only for nonstiff problems (they are explicit integrators). The development of stiff (implicit) integrators is considerably more involved than in the preceding development, so we merely consider in the next section why they might be required for a particular problem, Equations 1.6 to 1.17.

Program 1.7 calls the two MATLAB nonstiff solvers, ode23 and ode45, for solution of Equations 1.3 and 1.4, with the evaluation of the exact solution, Equation 1.5, to assess the accuracy of the numerical solution.

```matlab
% Program 1.7
% Tumor model of eqs. (1.47), (1.48)
% (or eqs. (1.3), (1.4), (1.5))
%
% Global variables
  global lambda alpha ncall;
```
% Model parameters
lambda=1.0;
alpha=1.0;
%
% Select method
for mf=1:2
%
% Error tolerances
reltol=1.0e-02;
abstol=1.0e-02;
for ncase=1:4
  reltol=1.0e-02*reltol;
  abstol=1.0e-02*abstol;
%
% Initialize counter for derivative evaluations
ncall=0;
%
% Variables for ODE integration
 t0=0.0;
 tf=10.0;
 tout=[t0:1.0:tf]';
 nout=11;
%
% Initial condition
y0=1.0;
%
% Call ODE integrator
 options=odeset('RelTol',reltol,'AbsTol',abstol);
 if (mf==1) [t,y]=ode23('ode1p7',tout,y0,options); end
 if (mf==2) [t,y]=ode45('ode1p7',tout,y0,options); end
%
% Display solution and error
fprintf('%10.0f %15.10f %15.10f %15.10f
',t(i),ye(i),y(i),erry(i));
end
fprintf('%10.0f',ncall);
%
Program 1.7
Program for the integration of Equation 1.48 by the library integrators ode23 and ode45

We can note the following points about Program 1.7:

• Three global variables are defined, which can then be shared between Program 1.7 and a function, ode1p7.m, called by Program 1.7 to define ODE (Equation 1.3). In other words, alpha, lambda, and ncall are used in function ode1p7.m, but their values are initialized in Program 1.7:

```
% Global variables
global lambda alpha ncall;
```

• lambda and alpha are then set numerically (and, again, these values will be available in ode1p7.m because they are global variables):

```
% Model parameters
lambda=1.0;
alpha=1.0;
```

ncall is initialized numerically later in the code.

• A method flag, mf, is set to one of two values: mf = 1 calls ode23 and mf = 2 calls ode45:

```
% Select method
for mf=1:2
```

• For each value of mf, four solutions are computed, with relative and absolute error tolerances of $10^{-4}$, $10^{-6}$, $10^{-8}$, and $10^{-10}$. ode23 and ode45 then attempt to adjust $h$ to meet these tolerances:
% Error tolerances
reltol=1.0e-02;
abstol=1.0e-02;
for ncase=1:4
    reltol=1.0e-02*reltol;
    abstol=1.0e-02*abstol;
end

At the beginning of each case (ncase = 1 to 4), a counter, ncall, is initialized that is then incremented each time the function ode1p7.m is called. This procedure gives the total number of calls to ode1p7.m and thus the number of derivative evaluations for each solution.

% Initialize counter for derivative evaluations
ncall=0;

Again, note that ncall is a global variable so its value is returned to Program 1.7.

The variables that define the interval in the independent variable, t, and when the solution is displayed are then initialized. tout is a vector containing t = 0, 1, ..., 10 (a total of 11 output values of t):

% Variables for ODE integration
t0=0.0;
tf=10.0;
tout=[t0:1.0:tf]';
nout=11;

Initial condition (Equation 1.4) is set to start the solution:

% Initial condition
y0=1.0;

The solution to Equation 1.3 is computed by ode12 or ode45, depending on the value of mf:

% Call ODE integrator
options=odeset('RelTol',reltol,'AbsTol',abstol);
if (mf==1) [t,y]=ode23('ode1p7',tout,y0,options); end
if (mf==2) [t,y]=ode45('ode1p7',tout,y0,options); end

Function option is first called to set the relative and absolute error tolerances. Function ode1p7.m defines ODE (Equation 1.3) by receiving the global variables alpha and lambda, and the current values of t and y to evaluate the right-hand side (RHS) of Equation 1.3:
function yt=ode1p7(t,y)
%
% Set global variables
    global lambda alpha ncall;
%
% ODE
    yt(1)=lambda*exp(-alpha*t)*y(1);
%
% Increment counter for derivative evaluations
    ncall=ncall+1;

Note that \( y \) is an input column vector (with one element, \( y(1) \), for the 1x1 ODE problem of Equation 1.3), and \( yt \) is an output column vector (with one element, \( yt(1) \)). \( t \) is an input scaler. Also, \( ncall \) is incremented by 1 each time \( ode1p7.m \) is called, which then provides the total number of calls (derivative evaluations) reported in the output.

- The parameters and numerical solution are then displayed by a series of \texttt{fprintf} statements, including the exact solution and the error in the solution:

\[
\begin{array}{c}
\text{% Display solution and error} \\
fprintf('mf = %1d
 case = %1d
 reltol = %6.2e
 abstol = %6.2e

 t ye y erry
');
for i=1:nout  \\
ye(i)=y0*exp((lambda/alpha)*(1.0-exp(-alpha*t(i))));
      erry(i)=ye(i)-y(i);
      fprintf('

 fprintf('

 end \\
fprintf('ncall = %5d
',ncall);

The first line of the first \texttt{fprintf} statement has been put on two lines to fit on the printed page, but would have to be returned to one line (since the line break \ldots cannot be used in the character string delimited by ‘\).

- After the four cases are completed for each of the two methods (a total of \( 4 \times 2 = 8 \) solutions), the final (eighth) solution is plotted via the \texttt{plot} and related statements and then saved via \texttt{print pro1p7.ps}. The resulting Postscript file is Figure 1.4; this plot is rather bumpy because only 11 output values of \( t \) (in vector \texttt{tout}) are used. Of course, this number could easily be increased.
Program 1.7, \( \frac{dy}{dt} = \lambda \exp(-\alpha t)y \)

FIGURE 1.4
Solution of Equations 1.3, 1.4, 1.5, from Program 1.7, \(mf = 2, ncase = 4\).

\[
\% \quad \% \text{Next case}
\text{end}
\%
\% \text{Next method}
\text{end}
\%
\% \text{Plot last solution}
\text{plot(t,y);}
\text{xlabel('t')}\n\text{ylabel('y(t)')}\n\text{title('Program 1.7,}
\text{dy/dt = } \lambda \exp(-\alpha t)y')\n\text{print pro1p7.ps}
\]

Note in the title statement that Greek letters can be included in the label in Figure 1.4 by using the codes \(\lambda\) and \(\alpha\).

- Finally, we should note that routine ode23 is based on a RK (2, 3) pair\(^3\) and ode45 is based on a RKF (4, 5) pair\(^3\). The error estimates in these two RK methods are used to adjust \(h\) to achieve the error tolerances specified in the call to function options. Some detailed coding for this type of error monitoring and control is given in the routines discussed in subsequent chapters.
The output from Program 1.7 is listed below:

```
mf = 1
case = 1
rehtol = 1.00e-004
abstol = 1.00e-004

<table>
<thead>
<tr>
<th>t</th>
<th>ye</th>
<th>y</th>
<th>erry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>1.0000</td>
<td>1.0000</td>
<td>0.000000000</td>
</tr>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8816</td>
<td>-0.0000002034</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3743</td>
<td>-0.0000638807</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.5864</td>
<td>-0.0000933201</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.6691</td>
<td>-0.0001103656</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>2.7002</td>
<td>-0.0001284242</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>2.7117</td>
<td>-0.0001434300</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>2.7160</td>
<td>-0.0001552112</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>2.7175</td>
<td>-0.0001581733</td>
</tr>
<tr>
<td>9.0</td>
<td>2.7179</td>
<td>2.7181</td>
<td>-0.0001592608</td>
</tr>
<tr>
<td>10.0</td>
<td>2.7182</td>
<td>2.7183</td>
<td>-0.0001592537</td>
</tr>
</tbody>
</table>

ncall = 73
```

```
mf = 1
case = 2
rehtol = 1.00e-006
abstol = 1.00e-006

<table>
<thead>
<tr>
<th>t</th>
<th>ye</th>
<th>y</th>
<th>erry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>1.0000</td>
<td>1.0000</td>
<td>0.000000000</td>
</tr>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8816</td>
<td>-0.0000003221</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3742</td>
<td>-0.00000012721</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.5863</td>
<td>-0.00000016231</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.6689</td>
<td>-0.00000018847</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>2.7000</td>
<td>-0.00000021190</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>2.7116</td>
<td>-0.00000023452</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>2.7158</td>
<td>-0.00000025176</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>2.7174</td>
<td>-0.00000026866</td>
</tr>
<tr>
<td>9.0</td>
<td>2.7179</td>
<td>2.7179</td>
<td>-0.00000028883</td>
</tr>
<tr>
<td>10.0</td>
<td>2.7182</td>
<td>2.7182</td>
<td>-0.00000029806</td>
</tr>
</tbody>
</table>

ncall = 268
```
mf = 1
case = 3
reltol = 1.00e-008
abstol = 1.00e-008

<table>
<thead>
<tr>
<th>t</th>
<th>ye</th>
<th>y</th>
<th>erry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>1.0000</td>
<td>1.0000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8816</td>
<td>-0.0000000072</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3742</td>
<td>-0.0000000188</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.5863</td>
<td>-0.0000000229</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.6689</td>
<td>-0.0000000260</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>2.7000</td>
<td>-0.0000000285</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>2.7116</td>
<td>-0.0000000309</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>2.7158</td>
<td>-0.0000000331</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>2.7174</td>
<td>-0.0000000354</td>
</tr>
<tr>
<td>9.0</td>
<td>2.7179</td>
<td>2.7179</td>
<td>-0.0000000374</td>
</tr>
<tr>
<td>10.0</td>
<td>2.7182</td>
<td>2.7182</td>
<td>-0.0000000394</td>
</tr>
</tbody>
</table>

ncall = 1180

mf = 1
case = 4
reltol = 1.00e-010
abstol = 1.00e-010

<table>
<thead>
<tr>
<th>t</th>
<th>ye</th>
<th>y</th>
<th>erry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>1.0000</td>
<td>1.0000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8816</td>
<td>-0.0000000001</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3742</td>
<td>-0.0000000003</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.5863</td>
<td>-0.0000000003</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.6689</td>
<td>-0.0000000003</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>2.7000</td>
<td>-0.0000000004</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>2.7116</td>
<td>-0.0000000004</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>2.7158</td>
<td>-0.0000000004</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>2.7174</td>
<td>-0.0000000004</td>
</tr>
<tr>
<td>9.0</td>
<td>2.7179</td>
<td>2.7179</td>
<td>-0.0000000005</td>
</tr>
<tr>
<td>10.0</td>
<td>2.7182</td>
<td>2.7182</td>
<td>-0.0000000005</td>
</tr>
</tbody>
</table>

ncall = 5392
mf = 2

case = 1
reltol = 1.00e-004
abstol = 1.00e-004

t    ye   y     erry
0.0  1.0000 1.0000 0.0000000000
1.0  1.8816 1.8815 0.00000785756
2.0  2.3742 2.3742 0.0000230677
3.0  2.5863 2.5862 0.0000132883
4.0  2.6689 2.6689 0.0000149230
5.0  2.7000 2.7000 0.0000165095
6.0  2.7116 2.7115 0.0000172423
7.0  2.7158 2.7158 0.0000175329
8.0  2.7174 2.7174 0.0000176427
9.0  2.7179 2.7179 0.0000176835
10.0 2.7182 2.7181 0.0000177068

ncall = 73

mf = 2

case = 2
reltol = 1.00e-006
abstol = 1.00e-006

t    ye   y     erry
0.0  1.0000 1.0000 0.0000000000
1.0  1.8816 1.8816 0.0000010053
2.0  2.3742 2.3742 0.0000010986
3.0  2.5863 2.5863 0.0000011560
4.0  2.6689 2.6689 0.0000010643
5.0  2.7000 2.7000 0.0000010083
6.0  2.7116 2.7116 0.0000009776
7.0  2.7158 2.7158 0.0000009652
8.0  2.7174 2.7174 0.0000009604
9.0  2.7179 2.7179 0.0000009586
10.0 2.7182 2.7182 0.0000009579

ncall = 85
mf = 2  
case = 3  
reltol = 1.00e-008  
abstol = 1.00e-008

<table>
<thead>
<tr>
<th>t</th>
<th>ye</th>
<th>y</th>
<th>err</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>1.0000</td>
<td>1.0000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8816</td>
<td>0.0000000237</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3742</td>
<td>0.0000000012</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.5863</td>
<td>0.0000000128</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.6689</td>
<td>0.0000001112</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>2.7000</td>
<td>0.0000000428</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>2.7116</td>
<td>0.000000092</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>2.7158</td>
<td>0.000000014</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>2.7174</td>
<td>-0.0000000231</td>
</tr>
<tr>
<td>9.0</td>
<td>2.7179</td>
<td>2.7179</td>
<td>-0.0000000081</td>
</tr>
<tr>
<td>10.0</td>
<td>2.7182</td>
<td>2.7182</td>
<td>0.000000077</td>
</tr>
</tbody>
</table>

ncall = 163

mf = 2  
case = 4  
reltol = 1.00e-010  
abstol = 1.00e-010

<table>
<thead>
<tr>
<th>t</th>
<th>ye</th>
<th>y</th>
<th>err</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>1.0000</td>
<td>1.0000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>1.0</td>
<td>1.8816</td>
<td>1.8816</td>
<td>0.0000000001</td>
</tr>
<tr>
<td>2.0</td>
<td>2.3742</td>
<td>2.3742</td>
<td>-0.0000000001</td>
</tr>
<tr>
<td>3.0</td>
<td>2.5863</td>
<td>2.5863</td>
<td>-0.0000000001</td>
</tr>
<tr>
<td>4.0</td>
<td>2.6689</td>
<td>2.6689</td>
<td>0.0000000003</td>
</tr>
<tr>
<td>5.0</td>
<td>2.7000</td>
<td>2.7000</td>
<td>0.0000000004</td>
</tr>
<tr>
<td>6.0</td>
<td>2.7116</td>
<td>2.7116</td>
<td>-0.0000000004</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7158</td>
<td>2.7158</td>
<td>0.0000000003</td>
</tr>
<tr>
<td>8.0</td>
<td>2.7174</td>
<td>2.7174</td>
<td>0.0000000002</td>
</tr>
<tr>
<td>9.0</td>
<td>2.7179</td>
<td>2.7179</td>
<td>-0.0000000003</td>
</tr>
<tr>
<td>10.0</td>
<td>2.7182</td>
<td>2.7182</td>
<td>0.0000000001</td>
</tr>
</tbody>
</table>

ncall = 385

We can note the following points about this output:

- Generally the output indicates that the relative error tolerances specified in the call to function *options* have been met. For example, for the first solution:
An error tolerance $\text{reltol} = 1.00e-004$ means that four figures of accuracy should be achieved in the numerical solution. In all cases, the numerical solution met this tolerance, e.g.,

$$
\begin{array}{cccc}
2.0 & 2.3742 & 2.3743 & -0.0000638807 \\
\end{array}
$$

indicates an error of $-0.000064$ or at least four figures in 2.3742. Similarly, $\text{absrel} = 1.00e-004$ indicates an absolute accuracy of 0.0001, and this was nearly achieved, e.g.,

$$
\begin{array}{cccc}
10.0 & 2.7182 & 2.7183 & -0.0001592537 \\
\end{array}
$$

or an absolute error of $-0.00016$. The same general conclusions apply to the other seven solutions. For example, with $\text{reltol} = 1.00e-010$, $\text{absrel} = 1.00e-010$, the solution was accurate to nearly ten figures for both $mf = 1$ and 2.

The apparent computational effort, as measured by the number of derivative evaluations (calls to function $ode1p7.m$) differed substantially between $ode23$ and $ode45$. 

```
mf = 1
    case = 4
    reltol = 1.00e-010
    abstol = 1.00e-010
    ncall = 5392
```
Thus, the calls to \textit{ode45} were less than $\frac{1}{10}$ those to \textit{ode23}. This result illustrates the relative efficiency of the higher-order method ((4, 5) is more efficient than (2, 3)). Note that this is true even though the (2, 3) pair requires the evaluation of $k_1$, $k_2$, and $k_3$ while the (4, 5) pair requires the evaluation of $k_1$, $k_2$, $k_3$, $k_4$, $k_5$, and $k_6$, i.e., each $k_i$ evaluation adds one derivative evaluation.

- For a general-purpose (library) ODE integrator, the coding for the problem and for the numerical integration algorithm should be separated. In this way, the coding for a new problem can be written, then combined with the coding for the algorithm (so that the algorithm coding remains unchanged). This is illustrated in Program 1.7 in which the problem ODE is defined in a function, in this case named \textit{ode1 p7.m}, and the algorithms are contained in \textit{ode23} and \textit{ode45}, which remain unchanged from one problem to the next. We shall use this division between problem-specific and general coding in the library routines to be considered subsequently. Note that this division was not used in Programs 1.1 to 1.6; in this sense, Programs 1.3 and 1.6 are the worst examples in that the ODE RHS (of Equation 1.3) was coded repeatedly (each time a $k_i$ was computed) rather than coding it just once as in function \textit{ode1 p7.m} called by Program 1.7. While this repetitive coding of the ODE in Programs 1.1 to 1.6 was not too cumbersome, we can imagine what it would be like, for example, for a 1000x1000 ODE system, with all 1000 ODEs programmed for each $k_i$!

We should also consider briefly the choice of error tolerances (the indiscriminate choice of error tolerances is probably the single most common reason for the failure of numerical library routines such as \textit{ode23} and \textit{ode45}). In this case, there was only one dependent variable, $y(t)$ defined by Equations 1.3 and 1.4. Further, since the range of values of $y$ was approximately $1 \leq y \leq 3$, the choice of the same value for the relative and absolute tolerances was reasonable (as suggested by the preceding discussion of the tolerances and the resulting accuracy of the solutions).

However, the selection of a single tolerance for both the relative and absolute errors, or even the same absolute error tolerance for a problem with more than one dependent variable, is not always appropriate. For example, if we are interested in solving a 2x2 problem, with $y_1$ having a typical value of 1000 (perhaps a temperature) and $y_2$ having a typical value of 0.01 (perhaps a concentration), we might select an absolute error tolerance of 0.1 for $y_1$ (1 part in 10,000), but this would be entirely too large for $y_2$ (10 parts in 1), and...
would result in a meaningless numerical solution for $y_2$, and most likely for $y_1$
as well since the ODEs for $y_1$ and $y_2$ would most certainly be connected, i.e.,

$$\frac{dy_1}{dt} = f_1(y_1, y_2, t)$$

$$\frac{dy_2}{dt} = f_2(y_1, y_2, t)$$

On the other hand, if we select an error tolerance of 0.000001 for $y_2$ (1 part in
10,000), this would be excessively small for $y_1$ (1 part in 1,000,000,000), and
would probably result in an excessively long computer run as the method
tried to adjust $h$ to meet this overly stringent error tolerance.

The solution to this situation might appear to be to select a relative error
tolerance such as 0.0001 (0.01% accuracy). However, a relative error is mean-
ingful only if the corresponding dependent variables are not zero anywhere
along the solution (but the absolute error criterion would not fail at such
points). Thus, some care might also have to be given to the selection of a
relative error. In general, the specification of both a relative tolerance and an
absolute tolerance might avoid problems with error monitoring and control
(automatic selection of $h$), but, again, different absolute tolerances might have
to be selected for different dependent variables, and even different relative
tolerances might also have to be selected for different dependent variables (de-
dpending on the sensitivity of the solution accuracy to the choice of the relative
tolerance). In general, the library ODE integration routines to be considered
subsequently will permit the selection of different relative and absolute tol-
erances for each dependent variable (but, again, appropriate values have to
be selected for each dependent variable, and indiscriminate choices without
much thought can lead to integrator failures, i.e., the failure to compute a
solution with acceptable accuracy, or to even compute any solution).

Parenthetically, function options will accept a vector for abstol and thus de-
fine an absolute error tolerance for each dependent variable (for the reason
explained with the preceding illustration of $y_1$ and $y_2$ having typical values
of 1000 and 0.01). However, reltol defined by a call to options will accept only
a scalar, so the same relative error tolerance is applied to all of the dependent
variables.

To conclude this section, we include Program 1.8 for the 2x2 problem of
Equations 1.6, 1.16, and 1.17, primarily to illustrate how ode23 and ode45 are
used for a problem with more than one ODE. Program 1.8 is listed below:

```matlab
% Program 1.8
% 2 x 2 system of eqs. (1.6), (1.16), (1.17)
%
% Global variables
  global a b;
```

Copyright 2004 by Chapman & Hall/CRC
% Model parameters
a=5.5;
b=4.5;
%
% Select method
for mf=1:2
%
% Error tolerances
reltol=1.0e-02;
abstol=1.0e-02;
for ncase=1:4
  reltol=1.0e-02*reltol;
  abstol=1.0e-02*abstol;
%
% Variables for ODE integration
t0=0.0;
tf=10.0;
tout=[t0:1.0:tf]';
nout=11;
%
% Initial condition
y10=0.0;
y20=2.0;
y0=[y10 y20]'
%
% Call ODE integrator
options=odeset('RelTol',reltol,'AbsTol',abstol);
if(mf==1)[t,y]=ode23('ode1p8',tout,y0,options); end
if(mf==2)[t,y]=ode45('ode1p8',tout,y0,options); end
%
% Display solution and error
fprintf('

 mf = %1d
 case = %1d
 reltol = %6.2e
 abstol = %6.2e

','mf',ncase,reltol,abstol);
fprintf(' t y1e y1 erry1
 y2e y2 erry2
');
for i=1:nout
  lambda1=-(a-b);
  lambda2=-(a+b);
  exp1=exp(lambda1*t(i));
  exp2=exp(lambda2*t(i));
y1e=(y10+y20)/2.0*exp1-(y20-y10)/2.0*exp2;
y2e=(y10+y20)/2.0*exp1+(y20-y10)/2.0*exp2;
  erry1=y1e-y(i,1);
  erry2=y2e-y(i,2);
Program 1.8

Program for the integration of Equations 1.6, 1.16, and 1.17 by the library integrators ode23 and ode45

We can note the following points about Program 1.8:

• The constants \( a \) and \( b \) in Equations 1.6, 1.16, and 1.17 are declared as global, then assigned numerical values:

\[
\begin{align*}
% & \quad \text{Global variables} \\
& \quad \text{global } a \ b; \\
& \quad \text{Model parameters} \\
& \quad a=5.5; \\
& \quad b=4.5;
\end{align*}
\]

• As in Program 1.7, two methods are used (\( mf = 1 \) for ode23 and \( mf = 2 \) for ode45). For each of these methods, a set of four error tolerances is used (again, these tolerances are appropriate for \( y_1 \) and \( y_2 \) since these variables range over (approximately) \( 0 \leq y_1, y_2 \leq 2 \)).

\[
\begin{align*}
% & \quad \text{Select method} \\
& \quad \text{for } mf=1:2 \\
% & \quad \text{Error tolerances} \\
& \quad \text{reltol}=1.0e-02;
\end{align*}
\]
abstol=1.0e-02;
for ncase=1:4
    reltol=1.0e-02*reltol;
    abstol=1.0e-02*abstol;
end

- The variables controlling the integration are the same as in Program 1.7:

    %
    % Variables for ODE integration
    t0=0.0;
    tf=10.0;
    tout=[t0:1.0:tf]';
    nout=11;

Thus, the $t$ scales for Programs 1.7 and 1.8 are the same, $0 \leq t \leq 10$, but clearly the $t$ scale is problem dependent, and thus a final value of $t = t_f$ must generally be selected for each new initial value problem. In other words, we must select $t_f$ to be large enough to encompass the entire solution, but not too large so that the essential details of the solution are confined to a small interval in $t$ (generally at the beginning of the solution). The selection of an appropriate $t$ scale is particularly important for stiff ODEs, as we shall observe in the next section on stability.

- The initial condition is now set as a vector (with two components):

    %
    % Initial condition
    y10=0.0;
    y20=2.0;
    y0=[y10 y20]';

Note that the last statement converts a row vector to a column vector (through the transpose operator, ') since an initial condition column vector is required by ode23 and ode45.

- ode23 and ode45 are called in the same way as in Program 1.7:

    %
    % Call ODE integrator
    options=odeset('RelTol',reltol,'AbsTol',abstol);
    if (mf==1)[t,y]=ode23('ode1p8',tout,y0,options); end
    if (mf==2)[t,y]=ode45('ode1p8',tout,y0,options); end

The only difference is that function ode1p8 is used to define the ODEs, Equations 1.6 and 1.16.

function yt=ode1p8(t,y)
%
% Set global variables
    global a b;
% ODEs
yt(1)=-a*y(1)+b*y(2);
yt(2)= b*y(1)-a*y(2);
yt=yt';

We can note the following points about ode1p8.m:
— A vector of derivatives, yt, with two elements is computed according to the ODEs, Equations 1.6 and 1.16. In other words, the dependent variable vector, y, is an input to ode1p8.m (generated by the integrator, ode23 or ode45), and the derivative vector, yt, is the output from ode1p8.m.
— Note also that this output derivative vector must be a column vector (required by ode23 and ode45), so a transpose is taken at the end of ode1p8.m.
• The numerical and exact solutions for y1 and y2 are then displayed (again, the character strings are put on two lines so that they fit on a printed page).

% Display solution and error
fprintf('

 mf = %1d
 case = %1d
 reltol = %6.2e
 abstol = %6.2e

',mf,ncase,reltol,abstol);
fprintf(' t y1e y1 erry1
 y2e y2 erry2
');
for i=1:nout
lambda1=-(a-b);
lambda2=-(a+b);
exp1=exp(lambda1*t(i));
exp2=exp(lambda2*t(i));
y1e=(y10+y20)/2.0*exp1-(y20-y10)/2.0*exp2;
y2e=(y10+y20)/2.0*exp1+(y20-y10)/2.0*exp2;
erry1=y1e-y(i,1);
erry2=y2e-y(i,2);
fprintf('%5.1f%9.4f%9.4f%15.10f
 %9.4f%9.4f%15.10f
',t(i),y1e,y(i,1),erry1,y2e,y(i,2),erry2);
end

In computing the exact solutions, the two eigenvalues $\lambda_1$ and $\lambda_2$ of Equations 1.16 are first computed ($\lambda$1 and $\lambda$2). The exponentials in Equations 1.17 corresponding to these eigenvalues are then computed ($\exp{1}$ and $\exp{2}$). Finally, the exact analytical solutions,
Equations 1.17, are programmed ($y_1e$ and $y_2e$), and then the corresponding truncation errors, $erry_1$ and $erry_2$, are computed.

- Note also that $ode23$ and $ode45$ actually return a matrix as the solution, $y$, consisting of $nout$ rows (for the $nout$ values of $t$), and two columns (for the two dependent variables, $y_1$ and $y_2$). This matrix is then used in the output lines:

```matlab
erry1=y1e-y(i,1);
erry2=y2e-y(i,2);
fprintf('%5.1f%9.4f%9.4f%15.10f\n
%9.4f%9.4f%15.10f\n\n',
    t(i),y1e,y(i,1),erry1,y2e,y(i,2),erry2);
```

where $i$ is set by the `for` statement

```matlab
for i=1:nout
```

- After the two `for` loops (which set $mf$ and $ncase$) are finished, the solution is plotted (corresponding to $mf = 2$ and $ncase = 4$):

```matlab
plot(t,y);
xlabel('t')
ylabel(['y1(t),y2(t)'])
title(' Program 1.8, 2 x 2 Linear System')
gtext('y1(t)');
gtext('y2(t)');
print pro1p8.ps
```

Note that function `plot` is able to accept the matrix $y$ directly since it checks for the correct number of rows in $t$ (a column vector with $nout$ rows). The resulting plot is then written to the Postscript file `pro1p8.ps` for storage.

The output from Program 1.8 is abbreviated below (to avoid excessive printed output) for $mf = 1$ and 2, $ncase = 1$ and 4:
mf = 1
case = 1
reltol = 1.00e-004
abstol = 1.00e-004

<table>
<thead>
<tr>
<th>t</th>
<th>y1e</th>
<th>y1</th>
<th>erry1</th>
<th>y2e</th>
<th>y2</th>
<th>erry2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000000000</td>
<td>2.0000</td>
<td>2.0000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>1.0</td>
<td>0.3678</td>
<td>0.3679</td>
<td>-0.0000179424</td>
<td>0.3679</td>
<td>0.3679</td>
<td>0.00000643053</td>
</tr>
<tr>
<td>2.0</td>
<td>0.1353</td>
<td>0.1353</td>
<td>0.00000767180</td>
<td>0.1353</td>
<td>0.1353</td>
<td>0.0000769485</td>
</tr>
<tr>
<td>3.0</td>
<td>0.0498</td>
<td>0.0497</td>
<td>0.00001011036</td>
<td>0.0498</td>
<td>0.0497</td>
<td>0.0000052134</td>
</tr>
<tr>
<td>4.0</td>
<td>0.0183</td>
<td>0.0182</td>
<td>0.0000911283</td>
<td>0.0183</td>
<td>0.0182</td>
<td>0.0000835132</td>
</tr>
<tr>
<td>5.0</td>
<td>0.0067</td>
<td>0.0068</td>
<td>-0.0000336414</td>
<td>0.0067</td>
<td>0.0066</td>
<td>0.0001147145</td>
</tr>
<tr>
<td>6.0</td>
<td>0.0025</td>
<td>0.0026</td>
<td>-0.00001634482</td>
<td>0.0025</td>
<td>0.0023</td>
<td>0.0001981412</td>
</tr>
<tr>
<td>7.0</td>
<td>0.0009</td>
<td>0.0010</td>
<td>-0.00001346866</td>
<td>0.0009</td>
<td>0.0008</td>
<td>0.0001489110</td>
</tr>
<tr>
<td>8.0</td>
<td>0.0003</td>
<td>0.0004</td>
<td>-0.00001107129</td>
<td>0.0003</td>
<td>0.0002</td>
<td>0.0001164729</td>
</tr>
<tr>
<td>9.0</td>
<td>0.0001</td>
<td>0.0002</td>
<td>-0.00001063790</td>
<td>0.0001</td>
<td>0.0000</td>
<td>0.0001086973</td>
</tr>
<tr>
<td>10.0</td>
<td>0.0000</td>
<td>0.0001</td>
<td>-0.00000217537</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000226717</td>
</tr>
</tbody>
</table>
mf = 1  
case = 4  
reltol = 1.00e-010  
abstol = 1.00e-010

<table>
<thead>
<tr>
<th>t</th>
<th>yle</th>
<th>yl</th>
<th>erry1</th>
<th>y2e</th>
<th>y2</th>
<th>erry2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000000000</td>
<td>2.0000</td>
<td>2.0000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>1.0</td>
<td>0.3678</td>
<td>0.3678</td>
<td>0.0000000000</td>
<td>0.3679</td>
<td>0.3679</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>2.0</td>
<td>0.1353</td>
<td>0.1353</td>
<td>0.0000000001</td>
<td>0.1353</td>
<td>0.1353</td>
<td>0.0000000001</td>
</tr>
<tr>
<td>3.0</td>
<td>0.0498</td>
<td>0.0498</td>
<td>0.0000000001</td>
<td>0.0498</td>
<td>0.0498</td>
<td>0.0000000001</td>
</tr>
<tr>
<td>4.0</td>
<td>0.0183</td>
<td>0.0183</td>
<td>0.0000000001</td>
<td>0.0183</td>
<td>0.0183</td>
<td>0.0000000001</td>
</tr>
<tr>
<td>5.0</td>
<td>0.0067</td>
<td>0.0067</td>
<td>0.0000000001</td>
<td>0.0067</td>
<td>0.0067</td>
<td>0.0000000001</td>
</tr>
<tr>
<td>6.0</td>
<td>0.0025</td>
<td>0.0025</td>
<td>0.0000000001</td>
<td>0.0025</td>
<td>0.0025</td>
<td>0.0000000001</td>
</tr>
<tr>
<td>7.0</td>
<td>0.0009</td>
<td>0.0009</td>
<td>0.0000000001</td>
<td>0.0009</td>
<td>0.0009</td>
<td>0.0000000001</td>
</tr>
<tr>
<td>8.0</td>
<td>0.0003</td>
<td>0.0003</td>
<td>0.0000000001</td>
<td>0.0003</td>
<td>0.0003</td>
<td>0.0000000001</td>
</tr>
<tr>
<td>9.0</td>
<td>0.0001</td>
<td>0.0001</td>
<td>0.0000000001</td>
<td>0.0001</td>
<td>0.0001</td>
<td>0.0000000001</td>
</tr>
<tr>
<td>10.0</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000000001</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000000001</td>
</tr>
</tbody>
</table>
\[
\begin{array}{cccc}
mf & = 2 \\
\text{case} & = 1 \\
\text{reltol} & = 1.00e-004 \\
\text{abstol} & = 1.00e-004 \\
\hline
\text{t} & y1e & y1 & erry1 \\
& y2e & y2 & erry2 \\
0.0 & 0.0000 & 0.0000 & 0.0000000000 \\
& 2.0000 & 2.0000 & 0.0000000000 \\
1.0 & 0.3678 & 0.3678 & -0.0000030880 \\
& 0.3679 & 0.3679 & 0.0000032018 \\
2.0 & 0.1353 & 0.1353 & 0.0000102110 \\
& 0.1353 & 0.1353 & -0.0000087609 \\
3.0 & 0.0498 & 0.0498 & 0.0000169871 \\
& 0.0498 & 0.0498 & -0.0000174932 \\
4.0 & 0.0183 & 0.0183 & 0.0000136219 \\
& 0.0183 & 0.0183 & -0.0000139397 \\
5.0 & 0.0067 & 0.0068 & -0.0000481707 \\
& 0.0067 & 0.0067 & 0.0000480671 \\
6.0 & 0.0025 & 0.0025 & 0.0000142956 \\
& 0.0025 & 0.0025 & -0.0000143637 \\
7.0 & 0.0009 & 0.0009 & -0.0000255897 \\
& 0.0009 & 0.0009 & 0.0000255619 \\
8.0 & 0.0003 & 0.0003 & 0.0000202537 \\
& 0.0003 & 0.0004 & -0.0000202664 \\
9.0 & 0.0001 & 0.0001 & 0.0000139694 \\
& 0.0001 & 0.0001 & -0.0000139749 \\
10.0 & 0.0000 & 0.0000 & 0.0000145650 \\
& 0.0000 & 0.0001 & -0.0000145673 \\
\end{array}
\]
Generally we can conclude that the error monitoring and control (automatic $h$ adjustment) worked as expected for both $ode23$ and $ode45$. The plot for $mf = 2$, $ncaze = 4$ appears in Figure 1.5. We can note in Figure 1.5 the initial conditions $y_1(0) = 0$, $y_2(0) = 2$ of Equations 1.6; as the solution evolves, the

<table>
<thead>
<tr>
<th>t</th>
<th>yle</th>
<th>yl</th>
<th>erryl</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td></td>
<td>2.0000</td>
<td>2.0000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>1.0</td>
<td>0.3678</td>
<td>0.3678</td>
<td>0.0000000000</td>
</tr>
<tr>
<td></td>
<td>0.3679</td>
<td>0.3679</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>2.0</td>
<td>0.1353</td>
<td>0.1353</td>
<td>0.0000000000</td>
</tr>
<tr>
<td></td>
<td>0.1353</td>
<td>0.1353</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>3.0</td>
<td>0.0498</td>
<td>0.0498</td>
<td>0.0000000000</td>
</tr>
<tr>
<td></td>
<td>0.0498</td>
<td>0.0498</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>4.0</td>
<td>0.0183</td>
<td>0.0183</td>
<td>0.0000000000</td>
</tr>
<tr>
<td></td>
<td>0.0183</td>
<td>0.0183</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>5.0</td>
<td>0.0067</td>
<td>0.0067</td>
<td>0.0000000000</td>
</tr>
<tr>
<td></td>
<td>0.0067</td>
<td>0.0067</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>6.0</td>
<td>0.0025</td>
<td>0.0025</td>
<td>0.0000000000</td>
</tr>
<tr>
<td></td>
<td>0.0025</td>
<td>0.0025</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>7.0</td>
<td>0.0009</td>
<td>0.0009</td>
<td>0.0000000000</td>
</tr>
<tr>
<td></td>
<td>0.0009</td>
<td>0.0009</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>8.0</td>
<td>0.0003</td>
<td>0.0003</td>
<td>0.0000000000</td>
</tr>
<tr>
<td></td>
<td>0.0003</td>
<td>0.0003</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>9.0</td>
<td>0.0001</td>
<td>0.0001</td>
<td>0.0000000000</td>
</tr>
<tr>
<td></td>
<td>0.0001</td>
<td>0.0001</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>10.0</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td></td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000000000</td>
</tr>
</tbody>
</table>
two components, \( y_1(t) \), \( y_2(t) \) come together at approximately \( t = 1 \), then follow a common path. This is consistent with the analytical solution, Equation 1.17. At \( t = 1 \) the exponential \( e^{\lambda_2 t} = e^{-(a+b)t} = e^{-(5.5+4.5)t} = e^{-10t} \) decays to insignificance in both \( y_1(t) \) and \( y_2(t) \); the two solutions then decay according to the exponential \( e^{\lambda_1 t} = e^{-(a-b)t} = e^{-(5.5-4.5)t} = e^{-t} \), and we need to compute to \( t = 10 \) for this exponential to fully decay.

This is a common feature of the solution to simultaneous linear ODEs. In particular:

- There is an initial interval, e.g., \( 0 \leq t \leq 1 \), in which all of the exponentials (for all of the ODE eigenvalues) are significant. However, the exponentials with the largest eigenvalues, in this case \( \lambda_2 = -10 \) decay rapidly.
- If the eigenvalues are complex, then the magnitudes of their real parts determine how long this initial transient or boundary layer persists (and, of course, the real parts should all be negative for the ODE system to be stable; if any of the eigenvalues have positive real parts, the corresponding exponentials will grow with increasing \( t \)).
- Once this initial transient is past, the solutions decay according to the smallest eigenvalues, in this case \( \lambda_1 = -1 \), which defines the total \( t \) scale of the solution to be approximately \( 0 \leq t \leq 10 \) (as reflected in Figure 1.5).
• If the eigenvalues are widely separated (a stiff system), the initial transient determined by the largest eigenvalues will be very short (and thus will require many small integration steps for an accurate solution in this interval). However, the total length or scale of the solution in $t$ will be determined by the smallest eigenvalues, but small steps will still have to be taken throughout the entire solution to maintain stability with a nonstiff (explicit) integrator such as the ones we have considered so far. It is this combination of small $h$ to cover a large interval in $t$ that makes stiff ODE systems relatively difficult to solve numerically with a nonstiff integrator.

We now consider some of the stability properties of ODE integrators for stiff and nonstiff ODEs. In other words, in addition to the previous consideration of accuracy, typically in the form of the order conditions, e.g., $O(h^p)$, $p = 1, 2, 3, 4, 5$, we must also consider a second important limitation of numerical integration algorithms, their stability, which is discussed in the next, and final, section of this chapter.

1.7 Stability of RK Methods

So far, we have assumed that $h$ will somehow be selected (either manually as in Programs 1.1 to 1.6, or automatically as in Programs 1.7 and 1.8) so as to achieve a numerical ODE solution of acceptable accuracy. Thus accuracy, at least so far in this discussion, has determined $h$. However, there are situations for which $h$ must be reduced to a level that ensures a stable numerical solution, and this restriction on $h$ will occur at a smaller value of $h$ than that determined by accuracy. The class of problems for which stability limits $h$ is termed stiff.

We start the discussion of stability by considering the model ODE (Equation 1.22) $dy/dt = \lambda y$, $y(0) = y_0$, where we have chosen $\text{real}(\lambda), < 0$ so that the solution

$$y(t) = y_0 e^{\lambda t} \quad (1.52)$$

is stable (decays exponentially with $t$). If we apply the Euler method, Equation 1.19 or 1.28, to this system, starting at the initial condition $y(0) = y_0$,

$$y_1 = y_0 + \frac{dy_0}{dt}h = y_0 + (\lambda y_0)h = y_0(1 + \lambda h)$$

For the next step from $y_1$ to $y_2$

$$y_2 = y_1 + \frac{dy_1}{dt}h = y_1 + (\lambda y_1)h = y_1(1 + \lambda h) = y_0(1 + \lambda h)(1 + \lambda h) = y_0(1 + \lambda h)^2$$
In general, after \( n \) steps to go from \( y_0 \) to \( y_n \)

\[
y_n = y_0(1 + \lambda h)^n
\]  
(1.53)

We can consider how this Euler solution (Equation 1.53) compares with the exact solution, Equation 1.52:

\[
\lambda h = -2 \quad n \quad y_n \\
1 \quad y_1 = y_0(1 - 2) = -y_0 \\
2 \quad y_2 = y_1(1 - 2) = -y_0(-1) = y_0 \\
3 \quad y_3 = y_2(1 - 2) = y_0(-1) = -y_0, \ etc.
\]

\[
\lambda h = -3 \quad n \quad y_n \\
1 \quad y_1 = y_0(1 - 3) = -2y_0 \\
2 \quad y_2 = y_1(1 - 3) = -2y_0(-2) = 4y_0 \\
3 \quad y_3 = y_2(1 - 3) = 4y_0(-2) = -8y_0, \ etc.
\]

\[
\lambda h = -0.5 \quad n \quad y_n \\
1 \quad y_1 = y_0(1 - 0.5) = 0.5y_0 \\
2 \quad y_2 = y_1(1 - 0.5) = 0.5y_0(0.5) = 0.25y_0 \\
3 \quad y_3 = y_2(1 - 0.5) = 0.25y_0(0.5) = 0.125y_0, \ etc.
\]

When \(|\lambda h| = 2\) (i.e., \(\lambda h = -2\)), the solution oscillates between \(y_0\) and \(-y_0\) (when it should decay according to Equation 1.52). When \(|\lambda h| > 2\) (i.e., \(\lambda h = -3\)), the solution grows in amplitude from \(y_0\) to \(-2y_0\) to \(4y_0\), etc. (the solution is unstable). However, when \(|\lambda h| < 2\) (i.e., \(\lambda h = -0.5\)), the solution decays (and is therefore stable). Thus, \(|\lambda h| = 2\) is the stability limit of the Euler method when applied to this model problem (Equation 1.22).

Also, since the eigenvalues of ODEs can, in general, be complex, the stability criterion \(|\lambda h| = 2\) defines a circle in the complex plane with center at \((-1, i0)\) and unit radius as illustrated in Figure 1.6. Note, in particular, the stability interval \(-2 \leq \lambda h \leq 0\) along the negative real axis when \(\lambda\) is real and negative (corresponding to a stable solution from Equation 1.52), and \(h\) is positive (the case we have considered, although the previous integration methods are valid for negative \(h\) corresponding to integration in the direction of decreasing \(t\)).

We can establish the stability region of Figure 1.6 for the Euler method (the interior of the circle is the stable region) by plotting a series of points in the complex plane corresponding to a series of values of \(\lambda h\). Consider the stepping formula for the Euler method from point \(y_i\) to point \(y_{i+1}\), Equation 1.19. If the numerical solution from this stepping formulas is to be stable, we require

\[
\left| \frac{y_{i+1}}{y_i} \right| \leq 1
\]

In other words, the absolute value of the solution at \(i + 1\) should be less than or equal to the absolute value at \(i\) as, for example, in the exponential decay of Equation 1.52 (the ratio \(|y_{i+1}/y_i|\) is generally called the amplification factor or stability function, and it should be less than one for a stable solution).
For the Euler method applied to the model ODE \( \frac{dy}{dt} = \lambda y, y(0) = y_0 \) (Equation 1.22)

\[
y_{i+1} = y_i + \frac{d y_i}{dt} h = y_i + (\lambda y_i)h = y_i (1 + \lambda h)
\]

or

\[
\frac{y_{i+1}}{y_i} = 1 + (\lambda h)
\]

For this ratio to have an absolute value of 1, even for complex \( \lambda \), we require

\[
1 + (\lambda h) = e^{i\theta}
\]

\( e^{i\theta} \) is a complex variable with unit magnitude, i.e., \(|e^{i\theta}| = 1\). Since \( e^{i\theta} = \cos \theta + i \sin \theta \) (the Euler identity),

\[
1 + \text{Re}(\lambda h) + i \text{Im}(\lambda h) = \cos \theta + i \sin \theta
\]

so that

\[
\text{Re}(\lambda h) = \cos \theta - 1
\]
\[
\text{Im}(\lambda h) = \sin \theta
\]
We can determine the values of $\text{Re}(\lambda h)$ and $\text{Im}(\lambda h)$ for a selected sets of values of $\theta$

<table>
<thead>
<tr>
<th>$\theta$</th>
<th>$e^{i\theta}$</th>
<th>$\lambda h$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$1 + i0$</td>
<td>$0 + i0$</td>
</tr>
<tr>
<td>$\pi/4$</td>
<td>$1/\sqrt{2} + i(1/\sqrt{2})$</td>
<td>$(1/\sqrt{2} - 1) + i(1/\sqrt{2})$</td>
</tr>
<tr>
<td>$\pi/2$</td>
<td>$0 + i1$</td>
<td>$-1 + i1$</td>
</tr>
<tr>
<td>$3\pi/4$</td>
<td>$-1/\sqrt{2} + i(1/\sqrt{2})$</td>
<td>$(-1/\sqrt{2} - 1) + i(1/\sqrt{2})$</td>
</tr>
<tr>
<td>$\pi$</td>
<td>$-1 + i0$</td>
<td>$-2 + i0$</td>
</tr>
<tr>
<td>$5\pi/4$</td>
<td>$-1/\sqrt{2} - i(1/\sqrt{2})$</td>
<td>$(-1/\sqrt{2} + 1) - i(1/\sqrt{2})$</td>
</tr>
<tr>
<td>$3\pi/2$</td>
<td>$0 - i1$</td>
<td>$-1 - i1$</td>
</tr>
<tr>
<td>$7\pi/4$</td>
<td>$1/\sqrt{2} - i(1/\sqrt{2})$</td>
<td>$(1/\sqrt{2} - 1) - i(1/\sqrt{2})$</td>
</tr>
<tr>
<td>$2\pi$</td>
<td>$1 + i0$</td>
<td>$0 + i0$</td>
</tr>
</tbody>
</table>

If $\text{Re}(\lambda h)$ is plotted vs. $\text{Im}(\lambda h)$, the resulting figure is a circle, centered at $(-1, i0)$ with unit radius (see Figure 1.6). If $h$ is chosen so that the complex point $\lambda h$ falls outside the circle, the numerical solution will be unstable (since $|y_{i+1}/y_i| > 1$). Thus, a stability limit is placed on $h$ for the explicit Euler method.

Usually, the accuracy requirement will set the step $h$ to a value smaller than for $|\lambda h| = 2$, as discussed previously. However, there is an exception to this conclusion. Consider the 2x2 system of Equations 1.6 and the analytical solution, Equation 1.17, for the special case $y_1(0) = 0$, $y_2(0) = 2$

$$y_1(t) = e^{\lambda_1 t} - e^{\lambda_2 t} \quad (1.54a)$$
$$y_2(t) = e^{\lambda_1 t} + e^{\lambda_2 t} \quad (1.54b)$$

for which the eigenvalues are $\lambda_1 = -(a - b)$, $\lambda_2 = -(a + b)$ as noted previously (recall again how Equations 1.54 appear numerically in Figure 1.5 for $a = 5.5$, $b = 4.5$ and $\lambda_1 = -1$, $\lambda_2 = -10$).

We now consider some additional particular values for $a$ and $b$

| Values of $a$, $b$ | Values of $\lambda_1$, $\lambda_2$ | $|\lambda_2|/|\lambda_1|$ and Description |
|---------------------|-----------------------------------|---------------------------------|
| **Case 1**          |                                   |                                 |
| $a = 50.5$          | $\lambda_2 = -100$               | $|\lambda_2|/|\lambda_1| = 100$ nonstiff |
| $b = 49.5$          | $\lambda_1 = -1$                 |                                 |
| **Case 2**          |                                   |                                 |
| $a = 500.5$         | $\lambda_2 = -1000$              | $|\lambda_2|/|\lambda_1| = 1000$ moderately stiff |
| $b = 499.5$         | $\lambda_1 = -1$                 |                                 |
| **Case 3**          |                                   |                                 |
| $a = 500, 000.5$    | $\lambda_2 = -1, 000, 000$       | $|\lambda_2|/|\lambda_1| = 1, 000, 000$ stiff |
| $b = 499, 999.5$    | $\lambda_1 = -1$                 |                                 |
Consider the maximum Euler step for the stiff case. If \( \lambda_2 = -1, 000, 000, \) the maximum stable step is given by \( |\lambda h| = 2 \) or \( h = 2/1, 000, 000 \) = 0.000002. However, to compute a complete solution, we require a final \( t \) given approximately by \( \lambda_1 t \approx -10 \) (or \( t = 10 \) so that \( \exp(\lambda_1 t) = \exp(-10) \) has decayed to insignificance compared to the initial condition \( y_2(0) = 1 \)). Thus, we must take \( 10/0.000002 = 5 \times 10^8 \) steps! If this does not seem like a large number of steps, consider \( a = 500, 000, 000.5, b = 499, 999, 999.5 \) for which the ratio

\[
\frac{|\text{largest eigenvalue}|}{|\text{smallest eigenvalue}|} = \frac{|\lambda_{\text{max}}|}{|\lambda_{\text{min}}|} = \frac{|\lambda_2|}{|\lambda_1|} = 10^9
\]

and \( 5 \times 10^9 \) steps would be required to compute a complete solution (physical problems in which this stiffness ratio \( \frac{|\lambda_{\text{max}}|}{|\lambda_{\text{min}}|} \approx 10^{12} \) to \( 10^{15} \) are not unusual).

As an incidental point, note that the calculation of \( \lambda_1 \) requires a subtraction, \( \lambda_1 = -(a - b) \). If \( a \) and \( b \) are nearly equal, e.g., \( a = 500, 000.5, b = 499, 999.5 \), then this subtraction might be done with substantial error. For example, if the machine precision (often termed the machine epsilon or unit roundoff) is \( 10^{-7} \) (one part in \( 10^7 \)) corresponding to 32-bit arithmetic, this stiff ODE system could not be integrated numerically since the calculation of \( (a - b) \) requires a precision better than more than one part in \( 10^7 \). Although this is a heuristic argument, generally the conclusion is correct, i.e., stiff systems require a precision that is substantially better than set by \( \frac{|\lambda_{\text{max}}|}{|\lambda_{\text{min}}|} \approx 10^{12} \) (1, 000, 000 in the preceding example). As an example of available precision, the machine epsilon for MATLAB and Java is approximately \( 10^{-15} \) so that an ODE system with a stiffness ratio approaching \( 10^{15} \) can be accommodated with these systems.

Thus, if we require the solution to a system of stiff ODEs, we should not use the Euler method (because of the stability limit \( |\lambda h| = 2 \)). We might consider a higher-order method, e.g., the modified Euler method of Equation 1.29, the classical fourth-order RK method of Equations 1.50, but if we do a similar stability analysis, we arrive at a stability limit that is not much greater than for the Euler method. For example, consider application of the fourth-order RK of Equations 1.50 to the model ODE (Equation 1.22) \( dy/dt = \lambda y, y(0) = y_0, \) where we have chosen \( \text{real}(\lambda) < 0 \) so that the solution, Equation 1.52 (or \( y e^{\lambda h} \) for one step \( h \) as noted after Equation 1.22), is stable. The RK constants for the model problem are

\[
\begin{align*}
    k_1 &= f(y_i, t_i)h = \lambda y_i h \\
    k_2 &= f(y_i + k_1/2, t_i + h/2)h = \lambda(y_i + \lambda h y_i/2)h \\
    k_3 &= f(y_i + k_2/2, t_i + h/2)h = \lambda[y_i + \lambda(y_i + \lambda h y_i/2)h/2]h \\
    k_4 &= f(y_i + k_3, t_i + h)h = \lambda[y_i + \lambda y_i + \lambda(y_i + \lambda h y_i/2)h/2]h \\
\end{align*}
\]
and the stepping formula is

\[ y_{i+1} = y_i + (1/6)(k_1 + 2k_2 + 2k_3 + k_4) \]

\[ = y_i + (1/6)[\lambda hy_i + 2\lambda(y_i + \lambda hy_i/2)h + 2\lambda(y_i + \lambda hy_i/2)h/2]h \]

\[ + \lambda(y_i + \lambda y_i + \lambda(y_i + \lambda hy_i/2)h/2)h] \]

\[ = y_i[1 + (1/6)[(1 + 2 + 2 + 1)(\lambda h) + (1/6)(1 + 1 + 1)(\lambda h)^2 \]

\[ + (1/6)(1/2 + 1/2)(\lambda h)^3 + (1/6)(1/4)(\lambda h)^4]] \]

\[ = y_i(1 + (\lambda h)/1! + (\lambda h)^2/2! + (\lambda h)^3/3! + (\lambda h)^4/4!) \] (1.55a)

Since the exact solution to the model problem is \( y = y_0e^{\lambda t} \) for the distance \( t = h \), the solution changes from \( y_i \) to \( y_{i+1} \) according to the exact solution

\[ y_{i+1} = y_i e^{\lambda h} = y_i[1 + (\lambda h)/1! + (\lambda h)^2/2! + (\lambda h)^3/3! + (\lambda h)^4/4! + \cdots] \] (1.55b)

Thus, the RK stepping formula fits the Taylor series solution of the exact solution up to and including the \((\lambda h)^4\) term (up to and including the \((d^4y_i/dt^4)(h^4/4!)\) term), as expected (compare Equations 1.55). Also, although the exact solution, Equation 1.55b, is stable for \( \lambda < 0 \), the approximate solution, Equation 1.55a, is not necessarily stable (because of the truncation). In fact, we can chose \( h \) large enough to make the solution of Equation 1.55a unstable.

Again, if the numerical solution is to remain stable, we require \( |y_{i+1}/y_i| \leq 1 \). Thus, for the limiting value \( |y_{i+1}/y_i| = 1 \)

\[ |1 + (\lambda h)/1! + (\lambda h)^2/2! + (\lambda h)^3/3! + (\lambda h)^4/4!| = 1 \] (1.55c)

and we can consider what values of \( \lambda h \) will satisfy Equation 1.55c. Clearly \( \text{real}(\lambda h) < 0 \). If \( \lambda h = -2.785 \),

\[ 1 + (-2.785)/1! + (-2.785)^2/2! + (-2.785)^3/3! + (-2.785)^4/4! = 1 \]

\[ 1 - 2.785 + 3.878 - 3.600 + 2.507 = 0.9996 \approx 1 \]

Thus, in place of the stability criterion for the Euler method, \( |\lambda h| \leq 2 \), we have for the fourth-order RK method along the negative real axis \( |\lambda h| = 2.785 \). In other words, the stability is not improved very much by going to a higher-order RK (or, in general, an explicit) method. For example, for the stiff case \( a = 500, 000.5, b = 499, 999.5 \) considered previously, for which the maximum stable step for the Euler method was \( h = 0.000002 \), the maximum stable step is now \( h = 0.000002785 \), and the required number of steps for a complete solution is reduced from \( 10/0.000002 = 5.0 \times 10^6 \) to \( 10/0.0000002785 = 3.59 \times 10^6 \) (not a very significant reduction, particularly when we recall that the classical fourth-order RK of Equations 1.50 requires four derivative evaluations for step \( h \) while the Euler method requires only one derivative evaluation).
We could also repeat the previous calculations for the Euler method to determine the stability region for the fourth-order RK method that is analogous to the circle in Figure 1.6, that is, by finding the values of $\lambda h$ that satisfy

$$1 + (\lambda h)/! + (\lambda h)^2/2! + (\lambda h)^3/3! + (\lambda h)^4/4! = e^{i\theta} \quad (1.55d)$$

This calculation is a little more involved than for the Euler method, so we merely state that the resulting stability diagram is in Schiesser, p. 157; Fortran and MATLAB program to calculate the boundary of the stability region is available from the authors (W.E.S.).

In summary, any explicit method will have a stability limit similar to the Euler method. So we might logically pose the question, “How do we efficiently compute the solution to a stiff ODE system?” The answer is that we must use an implicit algorithm (which generally will have a much larger stability region than an explicit method). Another way we can understand the stability limit of the explicit Euler method, Equation 1.19 (in addition to the preceding analysis which led to $|\lambda h| = 2$), is to recall the projection along a straight line in Figure 1.1. Specifically, if any eigenvalue of an ODE system is large, the solution changes rapidly (with respect to $t$), and the projection will therefore be highly inaccurate in going from $y_i$ to $y_{i+1}$ unless $h$ is very small. One way to avoid this required use of a small $h$ is to evaluate the derivative $dy/dt$ at $i + 1$ rather than at $i$, i.e., to use the implicit Euler method

$$y_{i+1} = y_i + \frac{dy_{i+1}}{dt} h \quad (1.56)$$

To show that using Equation 1.56 will be effective for stiff ODEs, we can repeat the preceding stability analysis for the ODE $dy/dt = \lambda y$, $y(0) = y_0$, where we have again chosen $\text{real}(\lambda) < 0$ so that the exact solution $y(t) = y_0 e^{\lambda t}$ is stable (decays exponentially with $t$). If we apply the implicit Euler method, Equation 1.56, to this system,

$$y_1 = y_0 + \frac{dy_1}{dt} h = y_0 + (\lambda y_1)h \text{ or } y_1/y_0 = \frac{1}{1 - \lambda h}$$

For the next step

$$y_2 = y_1 + \frac{dy_2}{dt} h = y_1 + (\lambda y_2)h \text{ or } \frac{y_2}{y_1} = \frac{1}{1 - \lambda h}$$

Then for two steps

$$\frac{y_2}{y_0} = \frac{y_2}{y_1} \frac{y_1}{y_0} = \left(\frac{1}{1 - \lambda h}\right) \left(\frac{1}{1 - \lambda h}\right) = \left(\frac{1}{1 - \lambda h}\right)^2 \quad (1.57)$$

In general, after $n$ steps

$$y_n = y_0 \left(\frac{1}{1 - \lambda h}\right)^n$$
We can again consider how this Euler solution (Equation 1.57) compares with the exact solution, \( y(t) = y_0 e^{\lambda t} \). A few specific values of the numerical solution are computed as before
\[
\lambda h = -2 \quad n \quad y_n \\
1 \quad y_1 = y_0 / (1 + 2) = y_0 / 3 \\
2 \quad y_2 = y_1 / (1 + 2) = y_0 (1/3)^2 \\
3 \quad y_3 = y_2 / (1 + 2) = y_0 (1/3)^3 
\]
Note that the solution decays in contrast to the explicit Euler method. This decay will occur no matter how large \( h \) is (in fact, the decay is faster with increasing \( h \)), so the implicit Euler method has no limit on \( \lambda h \) with respect to stability, i.e., the implicit Euler method is unconditionally stable. The step \( h \) is therefore only limited by accuracy.

However, there is generally a price to be paid for the enhanced stability of implicit methods. If the model equation is
\[
\frac{dy}{dt} = f(y, t)
\]
and \( f(y, t) \) is nonlinear, then application of the implicit Euler method, Equation 1.56, gives
\[
y_{i+1} = y_i + \frac{dy_{i+1}}{dt} h = y_i + f(y_{i+1}, t_{i+1}) h
\]
Note that the solution at the advanced point, \( y_{i+1} \), now appears on both sides of the stepping formula, and we therefore must solve a nonlinear equation to compute \( y_{i+1} \). Thus, the price we pay in general when using implicit methods is the solution of systems of nonlinear (algebraic) equations. For stiff ODEs, the additional effort of solving systems of nonlinear equations is usually well worthwhile since much larger integration steps are possible because of the improved stability characteristics of implicit methods (for example, for the previous stiff problem with \( \lambda_1 = -1, \lambda_2 = -1, 000, 000 \), rather than \( 5 \times 10^6 \) required steps for the explicit Euler method, probably a few hundred steps would be sufficient with the implicit Euler method to achieve reasonable accuracy since stability is not an issue; thus orders-of-magnitude reductions in the number of integration steps can be achieved by using an implicit integrator). Conversely, using a stiff (implicit) integrator on a nonstiff problem will waste computer time since the solution of nonlinear equations is unnecessary (as we observed in Programs 1.1 to 1.6).

We concluded the implicit Euler method is unconditionally stable, but it also has low accuracy (it is first order). Thus, we seek integration algorithms that have good stability and good accuracy. A widely used implicit method for stiff ODEs that has good combined stability and accuracy is based on the backward differentiation formulas (BDFs), which have the following general
form (References 5 and 6, pp. 183–184). The BDF stepping formula is
\[ \alpha_0 y_{i+1} + \alpha_1 y_i + \cdots + \alpha_v y_{i-v+1} = hf(y_{i+1} t_{i+1}) \] (1.58)
for \( dy/dt = f(y, t) \), where \( v \) is the order of the method (that defines the coefficients in a particular row of the following table of coefficients)

<table>
<thead>
<tr>
<th>( \nu )</th>
<th>( \alpha_0 )</th>
<th>( \alpha_1 )</th>
<th>( \alpha_2 )</th>
<th>( \alpha_3 )</th>
<th>( \alpha_4 )</th>
<th>( \alpha_5 )</th>
<th>( \alpha_6 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>[ \frac{3}{2} ]</td>
<td>-2</td>
<td>[ \frac{1}{2} ]</td>
<td>[ \frac{1}{3} ]</td>
<td>[ \frac{1}{6} ]</td>
</tr>
<tr>
<td>2</td>
<td>[ \frac{11}{6} ]</td>
<td>-3</td>
<td>[ \frac{3}{2} ]</td>
<td>-1</td>
<td>[ \frac{3}{4} ]</td>
<td>[ \frac{1}{4} ]</td>
<td>[ \frac{1}{5} ]</td>
</tr>
<tr>
<td>3</td>
<td>[ \frac{25}{12} ]</td>
<td>-4</td>
<td>3</td>
<td>-[ \frac{4}{3} ]</td>
<td>[ \frac{1}{5} ]</td>
<td>[ \frac{1}{6} ]</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>[ \frac{137}{60} ]</td>
<td>-5</td>
<td>-6</td>
<td>-20/3</td>
<td>[ \frac{5}{4} ]</td>
<td>[ \frac{1}{5} ]</td>
<td>[ \frac{1}{6} ]</td>
</tr>
<tr>
<td>5</td>
<td>[ \frac{147}{60} ]</td>
<td>-6</td>
<td>15/2</td>
<td>-20/3</td>
<td>[ \frac{15}{4} ]</td>
<td>[ \frac{6}{5} ]</td>
<td>[ \frac{1}{6} ]</td>
</tr>
</tbody>
</table>

Note that the solution at the advanced point, \( y_{i+1} \), appears on both sides of the stepping formula, Equation 1.58, so that in general the calculation of \( y_{i+1} \) requires the solution of a nonlinear equation (if \( f(y, t) \) from the ODE is nonlinear), or systems of nonlinear algebraic equations for the \( n \times n \) ODE problem.

The first-order BDF method (\( \nu = 1 \)) is just the implicit Euler method, Equation 1.56; note the weighting coefficients \( \alpha_0 = 1, \alpha_1 = -1 \) from the preceding table for which Equation 1.58 can be written as
\[ \alpha_0 y_{i+1} + \alpha_1 y_i = hf(y_{i+1} t_{i+1}) \]

or
\[ (1) y_{i+1} + (-1)y_i = hf(y_{i+1} t_{i+1}) \]
which is the implicit Euler method, Equation 1.56.

As we noted previously, the implicit Euler method is stable over the entire left half of the complex plane. However, it has limited accuracy; recall from Figure 1.1 that the Euler method is based on a first-order polynomial (linear approximation to the ODE solution), which is the case for both the explicit and implicit Euler methods (both are \( O(h) \)). For \( \nu = 2, \ldots, 6 \), the BDF methods are based on second- to sixth-order polynomial approximations of the ODE solution, which accounts for their good accuracy.

The BDF methods are implemented in the MATLAB routines \texttt{ode23s} and \texttt{ode15s} (presumably the “s” in these names denotes “stiff”). State-of-the-art implementations of the BDFs are available in the routines LSODE, LSODES, and DASSL (References 4, 7, pp. 55–64, and 8) that vary both \( \nu \) and \( h \) automatically (termed \textit{variable order-variable step} implementations).

The stability properties of the BDFs are summarized by their stability diagrams, which are presented in Reference 4, p. 163 (Fortran and MATLAB programs for calculating the stability boundaries are available from the authors).
In particular, for $\nu = 1$, the unstable region is within a circle centered at $(1, i0)$ with unit radius. This follows from Equation 1.57 in the same way that we established the stability domain for the explicit Euler method from Equation 1.53. Specifically,

- If $\lambda h = (1, i0)$ (real 1), there will be a division by zero in Equation 1.57. This demonstrates why the circle of instability for the implicit Euler method is centered at $(1, 0i)$.
- As we move away from $(1, 0i)$, the instability remains until we reach the boundary of the circle where the implicit Euler method switches from unstable to stable (the amplification factor from Equation 1.57, $\frac{y_{i+1}}{y_i} = \left| \frac{1}{1 - \lambda h} \right|$ switches from $>1$ to $<1$).
- Since we are discussing the right half of the complex plane, for which $\lambda$ has a positive real part, the exact solution to the ODE is unstable (grows exponentially with increasing $t$). This is an unusual situation in most applications of ODEs (e.g., unstable physical systems); in other words, the left half of the complex plane (where the real parts of all the ODE eigenvalues are negative) is usually of primary interest in applications. For the left half plane, the implicit Euler method is unconditionally stable.
- For the BDF order $\nu > 2$, the BDF methods have a region along the imaginary axis in the left half plane for which these methods are unstable.
- This region of instability in the left half plane becomes larger with increasing $\nu$.

In summary, the entire left half of the complex plane is unconditionally stable for $\nu \leq 2$, but a portion of the left half plane along the imaginary axis is unstable for $\nu \geq 3$, and this unstable region increases in size with increasing $\nu$. Thus, stiff ODEs with complex eigenvalues that fall close to the imaginary axis may cause long computer runs due to the limited stability along the imaginary axis (for this case, limiting $\nu$ to 2 can often increase the computational efficiency substantially, even though the order is relatively low). All of the BDFs for $\nu \leq 6$ are unconditionally stable along the negative real axis. These properties are readily observed in the BDF stability diagrams cited previously.

We should also note that the BDF stepping formula, Equation 1.58, requires a series of values prior to $y_{i+1}$, i.e., $y_i, y_{i-1}, \ldots, y_{i-\nu+1}$. At the beginning of the solution, when only one value is available, the initial condition $y_0$, we cannot use Equation 1.58 unless we start with the first-order method corresponding to $\nu = 1$. This will permit the calculation of $y_1$ and now we have two past values to calculate $y_2$, etc. Thus, we must build up the required past values starting with the $\nu = 1$ method (the implicit Euler method). In other words, the BDFs for $\nu > 1$ are not self-starting.
On the other hand, the RK methods are self-starting since their stepping formulas involve just $y_i$ to calculate $y_{i+1}$. Thus, we can use RK methods to calculate the first several values required by the BDF methods. However, for stiff problems, this requires the use of implicit RK methods (which we have not considered in the preceding discussion). Implicit RK methods are available, and have been implemented in a quality code, RADAU5.\textsuperscript{9}

Finally, an extensive library of quality public domain scientific software is available from the Internet.\textsuperscript{10} This library includes the source code for an extensive set of ODE integrators, including LSODE, LSODES, and DASSL. Since stiff ODE systems are an important class of initial value ODE problems, additional discussion of stiff integrators is given in Appendix C. In particular,

- The $2 \times 2$ system of Equations 1.6 for the stiff case $a = 500,000.5, b = 499,999.5$ is integrated by a fixed-step prototype BDF integrator for $\nu = 1$ (implicit Euler method) using MATLAB. This prototype illustrates the solution of the nonlinear equations required by an implicit integrator using Newton’s method. MATLAB programs for $\nu = 2, 3$ are available from the authors (W.E.S.).
- The same $2 \times 2$ problem is integrated using ode23s and ode45s to demonstrate both the stability and accuracy of these integrators. These programs can easily be modified for application to other stiff ODE problems.

This completes the introduction to ODE integrators, principally for the explicit RK integrator pairs $(1, 2), (2, 3), (2, 4), (4, 5)$ that are implemented in the routines to be discussed in the remainder of this book. We start with the solution of the $1 \times 1$ ODE system of Equations 1.3, 1.4, and 1.5 using integrators that vary $h$ to achieve a prescribed accuracy, that is, variable step explicit RK integrators.

References


Copyright 2004 by Chapman & Hall/CRC
2

Solution of a 1 x 1 ODE System

In this and subsequent chapters, we consider the programming of (a) the two ODE problems discussed in Chapter 1 and (b) two PDE problems. We start in this chapter with the 1x1 ODE problem of Equations 1.3 and 1.4, with the analytical solution of Equation 1.5. For each problem, we discuss the programming in C, C++, Fortran, Java, Maple, and MATLAB (these programming languages are listed here in alphabetical order, but they generally will be considered in a different order in the subsequent discussion).

The intention in providing the solutions of the four test problems is to give the reader enough detail that the programming of other ODE/PDE problems should be facilitated, possibly by using the programs presented as starting points, i.e., templates that can be used for new problems. Also, the programming of a common problem in all six languages should facilitate the conversion from one language to another (through a comparison of the corresponding coding).

2.1 Programming in MATLAB

A main program for the numerical integration of Equation 1.3 subject to initial condition Equation 1.4 is listed below:

```
% Main program ode1x1 computes the numerical
% solution to the 1 x 1 ODE system by one of
% six integrators
%
% Set integration parameters
[neqn,nout,nsteps,int,t0,tf,abserr,relerr]=intpar;
```
% Initial condition vector
    [u0]=initial(neqn,t0);
%
% Output interval
    tp=tf-t0;
%
% Compute solution at nout output points
    for j=1:nout
        % Print current solution
            [out]=fprint(int,neqn,t0,u0);
        % Fixed step modified Euler integrator
            if int == 1
                [u]=euler2a(neqn,t0,tf,u0,nsteps);
            end
        % Variable step modified Euler integrator
            if int == 2
                [u]=euler2b(neqn,t0,tf,u0,nsteps,abserr,relerr);
            end
        % Fixed step classical fourth order RK integrator
            if int == 3
                [u]=rkc4a(neqn,t0,tf,u0,nsteps);
            end
        % Variable step classical fourth order RK integrator
            if int == 4
                [u]=rkc4b(neqn,t0,tf,u0,nsteps,abserr,relerr);
            end
        % Fixed step RK Fehlberg (RKF45) integrator
            if int == 5
                [u]=rkf45a(neqn,t0,tf,u0,nsteps);
            end
        % Variable step RK Fehlberg (RKF45) integrator
            if int == 6
                [u]=rkf45b(neqn,t0,tf,u0,nsteps,abserr,relerr);
            end
        % Advance solution
            t0=tf;
            tf=tf+tp;
\begin{verbatim}
    u0=u;
    \%
    \% Next output
    end
    \%
    \% End of ode1x1
\end{verbatim}

Program 2.1.1
MATLAB main program for the numerical integration of Equation 1.3 subject to initial condition Equation 1.4

We can note the following points about the preceding program:

- Function \textit{intpar} is called to set the parameters that control the numerical integration of Equation 1.3.
  \%
  \% Set integration parameters
  [neqn,nout,nsteps,int,t0,tf,abserr,relerr]=intpar;
  \textit{intpar} will be subsequently listed and discussed.
- Function \textit{inital} is then called to set initial condition Equation 1.4
  \%
  \% Initial condition vector
  [u0]=inital(neqn,t0);

\textit{neqn}, the number of first-order ODEs, in this case, 1 for Equation 1.3, was set by the call to \textit{intpar} and is an input to \textit{inital}; similarly, the initial value of the independent variable, \( t_0 \) was set to 0 (zero) by the call to \textit{intpar}.
\textit{inital} is listed below:

\begin{verbatim}
    function [u0]=inital(neqn,t)
    \%
    \% Function inital sets the initial condition vector
    \% for the 1 x 1 ODE problem
    \%
    u0(1)=1.0;
    \%
    \% End of inital
\end{verbatim}

Program 2.1.2
Function \textit{inital} for initial condition Equation 1.4

Note that the initial condition of Equation 1.4 is 1.0. Also, the initial value of the dependent variable is actually in a 1D array or a column vector \( u_0 \), with a subscript 1. Note that the array is automatically defined by MATLAB, and does not have to be sized in a declarative statement.
• The interval in the independent variable, \( tp \) for the output of the numerical solution is the difference between the final value of the independent variable, \( tf \), set to 1.0 in \( intpar \) and the initial value of the independent variable, \( t0 = 0.0 \).

\[
\%
\text{Output interval}
\]
\[
tp = tf - t0;
\]

• The numerical solution is then computed and displayed at \( nout \) output points, with \( nout \) set in \( intpar \) to 6:

\[
\%
\text{Compute solution at nout output points}
\]
\[
\text{for } j = 1 : nout
\]
\[
\%
\text{Print current solution}
\]
\[
[out] = fprint(int, neqn, t0, u0);
\]

At the beginning of each pass through this \textit{for} loop, function \textit{fprint} is called to display (output) the solution. For the first pass through the loop \((j = 1)\), the initial conditions set in \textit{initial} are displayed. Function \textit{fprint} is subsequently listed and displayed.

• Within the \textit{for} loop, one of six integrators is called using \textit{int}, which is set in \textit{intpar}. The values of \textit{int} and the corresponding integrators are listed below:

<table>
<thead>
<tr>
<th>int</th>
<th>Integration algorithm</th>
<th>Discussion in Chapter 1</th>
<th>MATLAB function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Fixed step modified Euler</td>
<td>Equations 1.26, 1.27, 1.28, 1.29, 1.30</td>
<td>euler2a</td>
</tr>
<tr>
<td>2</td>
<td>Variable step modified Euler</td>
<td>Discussion after Equations 1.26</td>
<td>euler2b</td>
</tr>
<tr>
<td>3</td>
<td>Fixed step classical fourth-order RK</td>
<td>Equations 1.50, 1.51, Section 1.5</td>
<td>rkc4a</td>
</tr>
<tr>
<td>4</td>
<td>Variable step classical fourth-order RK</td>
<td>Equations 1.50, 1.51, discussion after Equations 1.26</td>
<td>rkc4b</td>
</tr>
<tr>
<td>5</td>
<td>Fixed step Runge Kutta Fehlberg 45</td>
<td>Equations 1.49, Section 1.5</td>
<td>rkf45a</td>
</tr>
<tr>
<td>6</td>
<td>Variable step Runge Kutta Fehlberg 45</td>
<td>Equations 1.49, discussion after Equations 1.26</td>
<td>rkf45b</td>
</tr>
</tbody>
</table>
The calls to these six integrators, determined by the value of int, are listed below:

```matlab
% Fixed step modified Euler integrator
if int == 1
    [u]=euler2a(neqn,t0,tf,u0,nsteps);
end

% Variable step modified Euler integrator
if int == 2
    [u]=euler2b(neqn,t0,tf,u0,nsteps,abserr,relerr);
end

% Fixed step classical fourth order RK integrator
if int == 3
    [u]=rkc4a(neqn,t0,tf,u0,nsteps);
end

% Variable step classical fourth order RK integrator
if int == 4
    [u]=rkc4b(neqn,t0,tf,u0,nsteps,abserr,relerr);
end

% Fixed step RK Fehlberg (RKF45) integrator
if int == 5
    [u]=rkf45a(neqn,t0,tf,u0,nsteps);
end

% Variable step RK Fehlberg (RKF45) integrator
if int == 6
    [u]=rkf45b(neqn,t0,tf,u0,nsteps,abserr,relerr);
end
```

Each of these ODE integrators are listed and discussed subsequently.

- After an interval $t_p$ is covered by the numerical integration from one of the six integrators, the solution is used as the starting point (in effect, a new initial condition), for the next interval, $t_p$. Note that although $u0$ and $u$ are 1D arrays (vectors), MATLAB automatically handles the individual elements of these arrays, i.e., indexing of the elements of these arrays is not required.

```matlab
% Advance solution
t0=tf;
```
The `end` statement terminates the `for` loop (i.e., serves as the final statement in the loop to initiate the next pass through the loop). When all `nout` passes through the loop have been completed, the execution of `ode1x1` is complete (in this case, a total of six passes since `nout = 6`).

The setting of the integration control parameters in `intpar` has been covered to some extent in the preceding discussion. The remaining parameters that are set are indicated in the following listing of `intpar`:

```matlab
function [neqn,nout,nsteps,int,t0,tf,abserr,relerr]=intpar
    % Function intpar sets the parameters to control the integration of the 1 x 1 ODE system
    % Number of first order ODEs
    neqn=1;
    % Number of output points
    nout=6;
    % Maximum number of steps in the interval t0 to tf
    nsteps=100;
    % Integrator
    int=1;
    % Initial, final values of independent variable
    t0=0.0;
    tf=1.0;
    % Error tolerances
    abserr=1.0e-05;
    relerr=1.0e-05;
end
```

**Program 2.1.3**
Function `intpar` called in Program 2.1.1 to set the integration parameters
The code in *intpar* that was not previously discussed is as follows:

- The maximum number of steps that an integrator will take over the interval $t_0 \leq t \leq t_f$:
  ```matlab
  nsteps=100;
  ```

- Selection of integrator 1 (*euler2a*):
  ```matlab
  int=1;
  ```

- Specification of the integration error tolerances:
  ```matlab
  abserr=1.0e-05;
  relerr=1.0e-05;
  ```

All of these parameters are used to control the numerical integration of Equation 1.3 by serving as inputs to one of the six integrators (*euler2a* in this case since $int = 1$).

Function *fprint* is listed below:

```matlab
function [out]=fprint(ncase,neqn,t,u)

out=t;

u0=1.0;
alpha=1.0;
lambda=1.0;

if(t<=0.0)
  fprintf('

 euler2a integrator

');
end
```

Copyright © 2004 by Chapman & Hall/CRC
Program 2.1.4
Function print called by Program 2.1.1 to display the numerical and analytical solutions to Equations 1.3 and 1.4
We can note the following points about \textit{fprint}:

- The parameters for the analytical solution, Equation 1.5, are first set:

  ```matlab
  % Problem parameters
  u0=1.0;
  alpha=1.0;
  lambda=1.0;
  %
  % Heading
  fprintf(' t  u1  u1e  u1-u1e\n');
  %
  % A heading for the output is then displayed at \( t = 0 \):
  ```

  ```matlab
  if(t<=0.0)
    % Label for ODE integrator
    % Fixed step modified Euler
    if(ncase==1)
      fprintf('\n euler2a integrator\n');
    % Variable step modified Euler
    elseif(ncase==2)
      fprintf('\n euler2b integrator\n');
    % Fixed step classical fourth order RK
    elseif(ncase==3)
      fprintf('\n rkc4a integrator\n');
    % Variable step classical fourth order RK
    elseif(ncase==4)
      fprintf('\n rkc4b integrator\n');
    % Fixed step RK Fehlberg 45
    elseif(ncase==5)
      fprintf('\n rkf45a integrator\n');
    % Variable step RK Fehlberg 45
    elseif(ncase==6)
      fprintf('\n rkf45b integrator\n');
    end
    %
    % Heading
    fprintf(' t   u1   u1e   u1-u1e\n');
  ```
Basically, this heading displays the particular integrator used (through the value of \textit{int}) and identifies the tabular numerical output.

- The analytical solution, Equation 1.5, and the difference between the numerical and analytical solutions are then computed:

\begin{verbatim}
\% Numerical and analytical solution output
\% Analytical solution
ue(1)=u0*exp(lambda/alpha*(1.0-exp(-alpha*t)));
\% Difference between exact and numerical solutions
diff=u-ue;
\end{verbatim}

- Finally, these numerical results are displayed:

\begin{verbatim}
\% Display the numerical and exact solutions, and
\% their difference
fprintf('%10.2f %10.5f %10.5f %10.5f \n',
t,u,ue(1),diff);
\end{verbatim}

Function \textit{deriv} for Equation 1.3 is listed below in Program 2.1.7. The output from this code is listed below:

euler2a integrator

\begin{tabular}{cccc}
\hline
\textbf{t} & \textbf{ul} & \textbf{ule} & \textbf{ul-ule} \\
\hline
0.00 & 1.00000 & 1.00000 & 0.00000 \\
1.00 & 1.88160 & 1.88160 & 0.00000 \\
2.00 & 2.37421 & 2.37421 & 0.00000 \\
3.00 & 2.58627 & 2.58626 & 0.00001 \\
4.00 & 2.66895 & 2.66895 & 0.00001 \\
5.00 & 2.70004 & 2.70003 & 0.00001 \\
\hline
\end{tabular}

We note that the integrator \textit{euler2a} computed a solution accurate to about 0.00001 using \textit{nsteps} = 100 steps or an integration step of \textit{h} = 1/100 within each output interval (corresponding to \textit{nout} = 6 or \textit{t} = 0, 1, \ldots, 5). Note that in this case the error tolerances set in \textit{intpar}, i.e., \textit{abserr} = 1.0e–05, \textit{relerr} = 1.0e–05 were not used since a fixed step integration is performed in \textit{euler2a}.

If the variable step modified Euler method of \textit{euler2b} is used (\textit{int} = 2 set in \textit{intpar}), the following output is produced:
euler2b integrator

<table>
<thead>
<tr>
<th>t</th>
<th>ul</th>
<th>ule</th>
<th>ul-ule</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.00000</td>
<td>1.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>1.00</td>
<td>1.88159</td>
<td>1.88160</td>
<td>-0.00001</td>
</tr>
<tr>
<td>2.00</td>
<td>2.37421</td>
<td>2.37421</td>
<td>0.00000</td>
</tr>
<tr>
<td>3.00</td>
<td>2.58626</td>
<td>2.58626</td>
<td>0.00000</td>
</tr>
<tr>
<td>4.00</td>
<td>2.66895</td>
<td>2.66895</td>
<td>0.00000</td>
</tr>
<tr>
<td>5.00</td>
<td>2.70003</td>
<td>2.70003</td>
<td>0.00001</td>
</tr>
</tbody>
</table>

Note that the error criteria are satisfied by the variable step modified Euler integrator.

Similar results can be produced by using int = 3, 4, 5, 6 in intpar. For example, with int = 6, the following output from rkf45b results:

rkf45b integrator

<table>
<thead>
<tr>
<th>t</th>
<th>ul</th>
<th>ule</th>
<th>ul-ule</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.00000</td>
<td>1.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>1.00</td>
<td>1.88159</td>
<td>1.88160</td>
<td>-0.00001</td>
</tr>
<tr>
<td>2.00</td>
<td>2.37421</td>
<td>2.37421</td>
<td>0.00000</td>
</tr>
<tr>
<td>3.00</td>
<td>2.58626</td>
<td>2.58626</td>
<td>-0.00001</td>
</tr>
<tr>
<td>4.00</td>
<td>2.66894</td>
<td>2.66895</td>
<td>-0.00001</td>
</tr>
<tr>
<td>5.00</td>
<td>2.70002</td>
<td>2.70003</td>
<td>-0.00001</td>
</tr>
</tbody>
</table>

Again, the error tolerances specified in intpar are satisfied.

We now consider the coding in each of the six integrators. euler2a is listed below

function [u]=euler2a(neqn,t0,tf,u0,nsteps)
% Function euler2a computes an ODE solution by a fixed step modified Euler method for a series of points along the solution by repeatedly calling function sseuler for a single modified Euler step.
% Argument list
% neqn number of first order ODEs
% t0 initial value of independent variable
% tf final value of independent variable
% u0 initial condition vector of length neqn

Copyright 2004 by Chapman & Hall/CRC
% nsteps number of modified Euler steps
% u ODE solution vector of length neqn after
% nsteps steps
%
% Integration step
h=(tf-t0)/nsteps;
%
% nsteps modified Euler steps
for i=1:nsteps
%
% Modified Euler step
[t,u,e]=sseuler(neqn,t0,u0,h);
%
% Reset base point values for next modified Euler step
u0=u;
t0=t;
%
% Next modified Euler step
end
%
% End of euler2a

Program 2.1.5
Integrator euler2a

We can note the following points about euler2a

- The integration step, \( h \), is set using parameters \( t_0, tf \), \( nsteps \) set in intpar:

  %
  % Integration step
  h=(tf-t0)/nsteps;

- \( nsteps \) modified Euler steps of length \( h \) are taken within the for loop by calling function sseuler (single step modified Euler):

  %
  % nsteps modified Euler steps
  for i=1:nsteps
    %
    % Modified Euler step
    [t,u,e]=sseuler(neqn,t0,u0,h);

- The solution from sseuler becomes the initial point for the next modified Euler step:
% Reset base point values for next modified Euler step
u0=u;
t0=t;

Again, note the use of the MATLAB feature for the arrays u, u0 (no subscripting is required).

• The for loop is then terminated, and after nsteps = 100 passes, the modified Euler integration over the interval t0 ≤ t ≤ tf is complete. A return to the main program ode1x1 of Program 2.1.1 then initiates the next interval in the for loop for j = 1 : nout.

% Next modified Euler step
end
% nsteps modified Euler steps completed

Function sseuler is an implementation of the modified Euler method discussed in Chapter 1.

function [t,u,e] = sseuler(neqn,t0,u0,h)
%
% Function sseuler computes an ODE solution by the modified
% Euler method for one step along the solution (by calls to
% derv to define the ODE derivative vector). It also
% estimates the truncation error of the solution, and
% applies this estimate as a correction to the solution
% vector.
%
% Argument list
%
% neqn number of first order ODEs
% t0 initial value of independent variable
% u0 initial condition vector of length neqn
% h integration step
% t independent variable
% u ODE solution vector of length neqn after
% one modified Euler step
% e estimate of truncation error of the solu-
%
Derivative vector at initial (base) point
[ut0]=derv(neqn,t0,u0);

First order (Euler) step
u=u0+ut0*h;
t=t0+h;

Derivative at advance point
[ut]=derv(neqn,t,u);

Truncation error estimate
e=(ut-ut0)*h/2.0;

Second order (modified Euler) solution vector
u=u+e;

End of sseuler

Program 2.1.6
Integrator sseuler for a single modified Euler step

We can note the following points about sseuler:

• The derivative vector at the base point is computed by a call to function derv:

  % Derivative vector at initial (base) point
  % [ut0]=derv(neqn,t0,u0);

  derv, which sets the RHS of Equation 1.3, is listed below:
  function [ut]=derv(neqn,t,u)
    % Function derv computes the derivative vector
    % of the 1 x 1 ODE problem
    % Problem parameters
    alpha=1.0;
    lambda=1.0;
    %
    % Derivative vector
    ut(1)=lambda*exp(-alpha*t)*u(1);
    %
    % End of derv

Program 2.1.7
Function derv for the RHS of Equation 1.3

Copyright © 2004 by Chapman & Hall/CRC
Once the derivative at the base point is returned from \textit{derv}, it is used in a Euler step:

\begin{verbatim}

% First order (Euler) step
u=u0+ut0*h;
t=t0+h;

\end{verbatim}

This code is just an implementation of Equation 1.26a, where, again, we have used the array features of MATLAB in working with \(u, u_0, ut_0\) (which are all one-element arrays or vectors since we are considering a 1x1 ODE system).

After the solution is computed at the advanced point, the derivative vector is computed at the advanced point by a second call to \textit{derv}:

\begin{verbatim}

% Derivative at advance point
[ut]=derv(neqn,t,u);

\end{verbatim}

Note that the second-order modified Euler method requires two derivatives evaluations (calls to \textit{derv}). In general, higher-order integration algorithms require additional derivative evaluations (this is the computational “price” that is paid for the improved accuracy of higher-order algorithms); also the order of the method, for example, second order for the modified Euler method, and the number of derivative evaluations, in this case two calls to \textit{derv} do not necessarily have to be the same number; generally, the number of derivative evaluations will be equal to or greater than the order of the algorithm.

The derivative vectors at the base point and the advanced point are then used to estimate the integration (truncation) error according to Equation 1.26b:

\begin{verbatim}

% Truncation error estimate
e=(ut-ut0)*h/2.0;

\end{verbatim}

This error can be used in a variable step method to determine if the integration error is within the specified error tolerance; in the present case of \textit{euler2a}, the estimated error is not used for this purpose since \textit{euler2a} implements a fixed step modified Euler method with \(h\) set at the value discussed previously (and passed to \textit{sseuler} through an argument).

Finally, the estimated error is used to correct the first-order Euler solution to produce the second-order modified Euler solution according to Equation 1.26c:

\begin{verbatim}

% Second order (modified Euler) solution vector
u=u+e;

\end{verbatim}
To complete the discussion of this 1x1 problem, *print* is called by *euler2a* as discussed previously to display the solution at the initial point $t = t_0$ and at subsequent output points (at intervals of $tf = 1.0$ as set in *ode1x1* or $t = 0, 1, 2, \ldots, 5$).

We have observed by taking $nout = 100$ fixed Euler steps, the numerical solution agreed with the analytical solution to an accuracy equal to the error tolerance. This agreement happened, however, by taking $nout$ large enough to achieve the required accuracy. Generally, for ODE problems for which we do not have an analytical solution (usually the case—if we knew the analytical solution, we would not have to compute a numerical solution), we would like the ODE integration algorithm to set the step size $h$ to produce a numerical solution that meets the specified error tolerance. While this may seem impossible (it suggests we know the error, and therefore must have an analytical solution), as we discussed in Chapter 1, we can in fact implement a variable step algorithm that automatically adjusts the integration step to (one hopes) meet the error tolerance. The details for the modified Euler method are discussed in the five-step algorithm discussed after Equation 1.26c. We now consider the code for implementing this algorithm in *euler2b*:

```matlab
function [u]=euler2b(neqn,t0,tf,u0,nsteps,abserr,relerr)

% Function euler2b computes an ODE solution by a variable
% step modified Euler method for a series of points along
% the solution by repeatedly calling function sseuler for
% a single modified Euler step. The truncation error is
% estimated along the solution to adjust the integration
% step according to a specified error tolerance.

% Argument list
% neqn number of first order ODEs
% t0 initial value of independent variable
% tf final value of independent variable
% u0 initial condition vector of length neqn
% nsteps maximum number of modified Euler steps
% abserr absolute error tolerance
% relerr relative error tolerance
```

Copyright 2004 by Chapman & Hall/CRC
% u       ODE solution vector of length eqn after
% nsteds steps
%
% Initial integration step
h=(tf-t0)/8.0;
%
% Minimum allowable step
hmin=(tf-t0)/nsteds;
%
% Start integration
  t=t0;
%
% While independent variable is less than the final
% value, continue the integration
  while t <= tf*0.999
%
% If the next step along the solution will go past
% the final value of the independent variable, set
% the step to the remaining distance to the final
% value
  if t+h > tf  h=tf-t; end
%
% Single modified Euler step
  [t,u,e]=sseuler(neqn,t0,u0,h);
%
% Flag for the end of the integration
  nfin1=1;
%
% Check if any of the ODEs have violated the error
% criteria
  for i=1:neqn
    if abs(e(i)) > abs(u(i))*relerr+abserr
%
%      Error violation, so integration is not complete.
%      Reduce integration step because of error violation
%      and repeat integration from the base point
%      h=h/2.0; nfin1=0; break;
    end
  end
%
% If the current step is less than the minimum
% allowable step, set the step to the minimum
% allowable value and continue integration from
% new base point
  if h < hmin  h=hmin; nfin1=1; end
% If there is no error violation, continue the integration from new base point
if nfin1 == 1 u0=u; t0=t;
%
% Test if integration step can be increased
for i=1:neqn
   if abs(e(i)) > (abs(u(i))*relerr+abserr)/4.0
      % Integration step cannot be increased
      nfin1=0; break;
   end
end
%
% Increase integration step
if nfin1 == 1 h=h*2.0; end
%
% Continue for no error violation (nfin1=1)
end
%
% Continue while
end
%
% End of euler2b

Program 2.1.8
Integrator euler2b

*euler2b* shares several features with *euler2a*. However, it also has additional coding to implement variable stepping in the modified Euler method. We consider now the details of this additional coding:

- The initial integration step, $h$, is now set (arbitrarily) to $\frac{1}{8}$ of the total output interval $tf-t0$. Note that this initial integration interval is larger than the constant value set in *euler2a* ($h = (tf-t0)/nout, nout = 100$) because we anticipate that *euler2b* will generally reduce $h$ only when the error tolerance has been exceeded; in other words, there will be a variable value of $h$, and we anticipate its value will not be reduced to $h = (tf-t0)/nout, nout = 100$ in order to meet the error tolerance.

% Initial integration step
h=(tf-t0)/8.0;
%
% Minimum allowable step
hmin=(tf-t0)/nsteps;
% Start integration
  \texttt{t=t0;}

However, since \texttt{euler2b} is a library integrator that may be applied to a spectrum of problems with different characteristics, we should also provide a safeguard against having $h$ reduced to excessively small values as it is adjusted in an attempt to satisfy the error tolerance. Therefore, we also define a minimum allowable integration step, $h_{\text{min}}$, which is given the same value as in \texttt{euler2a}; this value is arbitrarily selected, and some other value could be used, especially if $h$ is reduced to $h_{\text{min}}$ frequently, in which case a smaller value of $h_{\text{min}}$ would probably be used. Then the independent variable is set to the initial value, $t = t_0$, to start the integration.

- The numerical integration continues until the independent variable $t$ reaches the final value, $t_f$. This advancement of the solution is controlled by the \texttt{while} statement:

  
  \begin{verbatim}
  \% While independent variable is less than the final value, continue the integration
  while \texttt{t <= tf*0.999}
  \%
  \% If the next step along the solution will go past the final value of the independent variable, set the step to the remaining distance to the final value
  \% if \texttt{t+h > tf} \texttt{h=tf-t;} \texttt{end}
  \end{verbatim}

  Also, if the remaining distance for $t$ to reach $t_f$ is less than the current step $h$, the step is reduced to $t_f - t$. This ensures that the final step taken by \texttt{euler2b} will be exactly to the next output point.

- \texttt{euler2b} then takes a modified Euler step as discussed previously by a call to \texttt{sseuler}:

  \begin{verbatim}
  \%
  \% Single modified Euler step
  \[t,u,e]=\texttt{sseuler(neqn,t0,u0,h);} \%
  \%
  \% Flag for the end of the integration
  \texttt{nfin1=1;}
  \end{verbatim}

  Since this step might take the solution to the final value $t = t_f$, a flag is set indicating that the solution may be complete, $\texttt{nfin1} = 1$.

- The estimated error $e$ returned by \texttt{sseuler} is then used to test if any of the dependent variables violated the error criterion:
Check if any of the ODEs have violated the error criteria

```
for i=1:neqn
    if abs(e(i)) > abs(u(i))*relerr+abserr
        % Error violation, so integration is not complete.
        % Reduce integration step because of error violation and repeat integration from the base point
        h=h/2.0; nfin1=0; break;
    end
end
```

Note in the error test that both the absolute and relative error tolerances, abserr and relerr, are used. If an error violation is detected, the integration step is halved and the flag indicating the end of the integration nfin1 is set to zero (the integration is not complete since it will be repeated with one half the previous integration step to improve the accuracy of the solution). Note that if any of the dependent variables violate the error criterion, the integration step is halved, i.e., the for loop is exited through the break. Conversely, if all of the dependent variables meet the error criterion, nfin1 remains at 1 indicating that so far, so good.

- If the preceding integration step reduction (halving) has reduced the integration step to or below the minimum allowable value, the integration step is set to the minimum allowable value, and the integration then proceeds using this minimum value.

```
% If the current step is less than the minimum allowable step, set the step to the minimum allowable value and continue integration from new base point
if h < hmin h=hmin; nfin1=1; end
```

The idea here is that the integrator has done the best it can (reduced h to hmin), and it therefore might as well continue. Of course, at this point an error message could be printed signaling that another solution should probably be computed with a smaller value for hmin for comparison with the current solution. This point illustrates an important advantage of having access to the source code for the integrator; additional coding and features can be added such as an error message warning the user of a possible error violation.

- If at this point nfin1 = 1, the integration can proceed by resetting the base solution values equal to the current solution values:
% If there is no error violation, continue the integration from new base point
if nfin1 == 1 u0=u; t0=t;

- Also, before the next step is taken along the solution, there is the possibility that the integration step could be increased. Thus, a test is made to determine if the estimated error for each dependent variable is less than \( \frac{1}{4} \) of the error tolerance. If so, the integration step is doubled before the next step along the solution is taken:

```matlab
% Test if integration step can be increased
for i=1:neqn
    if abs(e(i)) > (abs(u(i))*relerr+abserr)/4.0
        % Integration step cannot be increased
        nfin1=0; break;
    end
end
% Increase integration step
if nfin1 == 1 h=h*2.0; end
% Continue for no error violation (nfin1=1)
end
% Continue while
end
```

Note that if all of the dependent variables have estimated errors less than \( \frac{1}{4} \) of the error tolerance, the integration step is doubled. In other words, if any of the dependent variables have an estimated error exceeding \( \frac{1}{4} \) of the error tolerance, the integration step is not doubled. Also, the factor of \( \frac{1}{4} \) comes from the second-order characteristic of the modified Euler method, i.e., if \( \text{error} = ch^2 \), then doubling \( h \) will increase \( \text{error} \) by a factor of four (which would bring the error just up to the specified error tolerance if the estimated error is \( \frac{1}{4} \) of the error tolerance). If \( nfin1 = 1 \), the integration step is doubled, and the integration proceeds; if \( nfin1 = 0 \), the integration proceeds without doubling of the step. In either case, stepping continues until \( t \) reaches \( t_f \) according to the original \( \text{while} \).

As noted previously, changes and new features can be added to this basic integrator. These could include:
• Selection of an initial integration step according to an established algorithm
• Change in the minimum allowable step, $h_{\text{min}}$
• Application of absolute and relative error tolerances to each dependent variable (so that these error tolerances can be specified for each dependent variable)
• A more sophisticated algorithm for increasing the integration step (rather than just doubling)

All of these features have been used in various library integrators. However, we have used only basic methods to facilitate understanding the variable stepping in $\text{euler2b}$. As we observed in the numerical output for the 1x1 ODE problem, these methods worked quite well (the error tolerance was maintained), and we have found that generally this is the case.

Also, we see that it is possible to adjust the integration step automatically to achieve a prescribed accuracy in the numerical solution without knowing the analytical (exact) solution. This is possible because we are using an error estimate to adjust the integration step rather than the exact error (which generally is unknown). Later we study in a little more detail how $h$ varies for this 1x1 ODE system.

This procedure of using an error estimate can be applied to an nxn ODE system. Furthermore, it can be used with a higher-order ODE integrator. To illustrate how this can be done, we now consider the following:

1. A fixed step classical fourth-order Runge Kutta (RK) integrator, $\text{rkc4a}$, as defined by Equations 1.50 and 1.51.
2. A variable step classical fourth-order RK integrator, $\text{rkc4b}$, as defined by Equations 1.50 and 1.51.

$r\text{k4a}$ and $r\text{k4b}$ closely parallel $\text{euler2a}$ and $\text{euler2b}$, respectively. Therefore, in the following discussion, we emphasize only the small differences (generally additional code to increase the order from second to fourth).

$r\text{k4a}$ (called by setting $int = 3$ in function $\text{intpar}$ of Program 2.1.3) is listed first:

```matlab
function [u]=rkc4a(neqn,t0,tf,u0,nsteps)

% Function rkc4a computes an ODE solution by a fixed step
% classical fourth order RK method for a series of points
% along the solution by repeatedly calling function ssrkc4
% for a single classical fourth order RK step.
% % Argument list
```
% neqn   number of first order ODEs
% t0     initial value of independent variable
% tf     final value of independent variable
% u0     initial condition vector of length neqn
% nsteps number of rkc4 steps
% u      ODE solution vector of length neqn after
%        nsteps steps

% Integration step
h=(tf-t0)/nsteps;

% nsteps rkc4 steps
for i=1:nsteps
  % rkc4 step
  [t,u,e]=ssrkc4(neqn,t0,u0,h);
  % Reset base point values for next rkc4 step
  u0=u;
  t0=t;
end
% Next rkc4 step
% End of rkc4a

Program 2.1.9
Integrator rkc4a

We see that rkc4a is essentially identical to euler2a; the only difference is
a call to ssrkc4 (for a single step classical fourth-order RK step) in place of
sseuler.
ssrkc4 is listed below:

function [t,u,e] = ssrkc4(neqn,t0,u0,h)
% Function ssrkc4 computes an ODE solution by the classical
% fourth order RK method for one step along the solution
% (by calls to derv to define the ODE derivative vector).
% It also estimates the truncation error of the solution,
and applies this estimate as a correction to the solution vector.

Argument list

- neqn number of first order ODEs
- t0 initial value of independent variable
- u0 initial condition vector of length neqn
- h integration step
- t independent variable
- u ODE solution vector of length neqn after one rkc4 step
- e estimate of truncation error of the solution vector

Derivative vector at initial (base) point
[ut0]=derv(neqn,t0,u0);

k1, advance of dependent variable vector and independent variable for calculation of k2
k1=h*ut0;
u=u0+0.5*k1;
t=t0+0.5*h;

Derivative vector at new u, t
[ut]=derv(neqn,t,u);

k2, advance of dependent variable vector and independent variable for calculation of k3
k2=h*ut;
u=u0+0.5*k2;
t=t0+0.5*h;

Derivative vector at new u, t
[ut]=derv(neqn,t,u);

k3, advance of dependent variable vector and independent variable for calculation of k4
\[ k_3 = h \cdot u_t; \]
\[ u = u_0 + k_3; \]
\[ t = t_0 + h; \]
\%
\% Derivative vector at new u, t
\[ [u_t] = \text{deriv(neqn}, t, u); \]
\%
\% k4
\[ k_4 = h \cdot u_t; \]
\%
\% Second order step
\[ \text{sum2} = u_0 + k_2; \]
\%
\% Fourth order step
\[ \text{sum4} = u_0 + (1.0/6.0) \cdot (k_1 + 2.0 \cdot k_2 + 2.0 \cdot k_3 + k_4); \]
\[ t = t_0 + h; \]
\%
\% Truncation error estimate
\[ e = \text{sum4} - \text{sum2}; \]
\%
\% Fourth order solution vector (from (2,4) RK pair);
\% two ways to the same result are listed
\% u = sum2 + e;
\[ u = \text{sum4}; \]
\%
\% End of ssrkc4

Program 2.1.10
Integrator \texttt{ssrkc4} for a classical fourth-order RK step

\texttt{ssrkc4} is similar in structure to \texttt{sseuler}; the only essential difference is the coding of the classical fourth-order RK algorithm, defined by Equations 1.50 and 1.51, in place of the modified Euler method of Equations 1.26.

To explain this difference a little further,

- The integration starts with an evaluation of the derivative vector at the base point (as in \texttt{euler2a}):

\%
\% Derivative vector at initial (base) point
\[ [u_t_0] = \text{deriv(neqn}, t_0, u_0); \]

- From the derivative vector at the base point, the first RK "constant," \( k_1 \), can be calculated according to Equation 1.50a:

\%
\% k1, advance of dependent variable vector and
\% independent variable for calculation of k2
\[ k_1 = h \cdot u(t_0); \]
\[ u = u_0 + 0.5 \cdot k_1; \]
\[ t = t_0 + 0.5 \cdot h; \]

Note that at the same time, the dependent variable vector, \( u \), is advanced in preparation for the calculation of \( k_2 \). Note also that the RK constants are vectors with a dimension equal to the number of first-order ODEs, one in this case (for the 1x1 ODE problem).

- The derivative vector is then computed at the new value of \( u \) by a call to `derv`, and \( k_2 \) is calculated according to Equation 1.50b; \( u \) is then advanced in preparation for the calculation of \( k_3 \):

```matlab
% Derivative vector at new u, t
[ut] = derv(neqn, t, u);
% k2, advance of dependent variable vector and
% independent variable for calculation of k3
k2 = h * ut;
    u = u0 + 0.5 * k2;
    t = t0 + 0.5 * h;
```

- The derivative vector is again computed at the new value of \( u \) by a call to `derv`, and \( k_3 \) is calculated according to Equation 1.50c; \( u \) is then advanced in preparation for the calculation of \( k_4 \):

```matlab
% Derivative vector at new u, t
[ut] = derv(neqn, t, u);
% k3, advance of dependent variable vector and
% independent variable for calculation of k4
k3 = h * ut;
    u = u0 + k3;
    t = t0 + h;
```

- The derivative vector is again updated and \( k_4 \) is calculated according to Equation 1.50d:

```matlab
% Derivative vector at new u, t
[ut] = derv(neqn, t, u);
% k4
    k4 = h * ut;
```

- The second- and fourth-order solutions are computed according to Equations 1.40 and 1.50e, respectively:

```matlab
% The second- and fourth-order solutions are computed according to
% Equations 1.40 and 1.50e, respectively:
```
% Second order step
sum2=u0+k2;
%
% Fourth order step
sum4=u0+(1.0/6.0)*(k1+2.0*k2+2.0*k3+k4);
t=t0+h;

• The integration error is estimated as the difference between the second-
  and fourth-order numerical solution, according to Equation 1.51:
%
% Truncation error estimate
e=sum4-sum2;

• The solution vector $u$ is then returned as the fourth-order result:
%
% Fourth order solution vector (from (2,4) RK pair);
% two ways to the same result are listed
% u=sum2+e;
% u=sum4;

The output from $rkc4a$ (which calls $ssrkc4$) is not listed here to conserve some
space. However, it is similar to the output from $euler2a$, and can easily be
produced by setting $int = 3$ in function $(intpar)$.

$rkc4b$, the variable step version of the classical fourth-order RK method of
Equations 1.50 and 1.51, is listed below:

function [u]=rkc4b(neqn,t0,tf,u0,nsteps,abserr,relerr)
  % Function rkc4b computes an ODE solution by a variable
  % step classical fourth order RK method for a series of
  % points along the solution by repeatedly calling function
  % ssrkc4 for a single classical fourth order RK step. The
  % truncation error is estimated along the solution to
  % adjust the integration step according to a specified
  % error tolerance.
  %
  % Argument list
  %
  % neqn number of first order ODEs
  %
  % t0 initial value of independent variable
  %
  % tf final value of independent variable
  %
% u0     initial condition vector of length neqn
% nsteps maximum number of rkc4 steps
% abserr absolute error tolerance
% relerr relative error tolerance
% u     ODE solution vector of length neqn after
% nsteps steps

% Initial integration step
h=(tf-t0)/2.0;

% Minimum allowable step
hmin=(tf-t0)/nsteps;

% Start integration
 t=t0;

% While independent variable is less than the final
% value, continue the integration
while t <= tf*0.999

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

% Single rkc4 step
[t,u,e]=ssrkc4(neqn,t0,u0,h);

% Flag for the end of the integration
nfin1=1;

% Check if any of the ODEs have violated the error
% criteria
for i=1:neqn
  if abs(e(i)) > abs(u(i))*relerr+abserr

% Error violation, so integration is not complete.
% Reduce integration step because of error violation
% and repeat integration from the base point
  h=h/2.0; nfin1=0; break;
end
end

% If the current step is less than the minimum allowable step, set the step to the minimum allowable value and continue integration from new base point
if h < hmin h=hmin; nfin1=1; end
%
% If there is no error violation, continue the integration from new base point
if nfin1 == 1 u0=u; t0=t;
%
% Test if integration step can be increased
for i=1:neqn
  if abs(e(i)) > (abs(u(i))*relerr+abserr)/16.0
%
    Integration step cannot be increased
    nfin1=0; break;
  end
end
%
% Increase integration step
if nfin1 == 1 h=h*2.0; end
%
% Continue for no error violation (nfin1=1)
end
%
% Continue while
end
%
% End of rkc4b

Program 2.1.11
Integrator rkc4b

rkc4b is basically identical to euler2b. The only essential difference is the call to ssrkc4 in place of sseuler. Also, the test for integration step doubling is

if abs(e(i)) > (abs(u(i))*relerr+abserr)/16.0

i.e., there is a division by 16 rather than by 4 as in sseuler since the classical RK method is fourth order (rather than second order).

To complete the picture, we include listings of the following:

- rkf45a (analogous to euler2a and rkc4a) as Program 2.1.12
- rkf45b (analogous to euler2b and rkc4b) as Program 2.1.13
- ssrkf45 (analogous to sseuler and ssrkc4) as Program 2.1.14
function [u]=rkf45a(neqn,t0,tf,u0,nsteps)

% Function rkf45a computes an ODE solution by the fixed
% step RK Fehlberg 45 method for a series of points along
% the solution by repeatedly calling function ssrkf45 for
% a single RK Fehlberg 45 step.
%
% Argument list
%
% neqn number of first order ODEs
% t0 initial value of independent variable
% tf final value of independent variable
% u0 initial condition vector of length neqn
% nsteps number of rkf45 steps
% u ODE solution vector of length neqn after
% nsteps steps
%
% Integration step
h=(tf-t0)/nsteps;
%
% nsteps rkf45 steps
for i=1:nsteps
%
% rkf45 step
[t,u,e]=ssrkf45(neqn,t0,u0,h);
%
% Reset base point values for next rkc4 step
u0=u;
t0=t;
%
% Next rkf45 step
end
%
% End of rkf45a

Program 2.1.12
Integrator rkf45a

rkf45a is called by setting int = 5 in function intpar. The output is essentially
the same as from euler2a and rkc4a, and therefore is not listed here.
function [u]=rkf45b(neqn,t0,tf,u0,nsteps,abserr,relerr)

% Function rkf45b computes an ODE solution by the variable
% step RK Fehlberg 45 method for a series of points along
% the solution by repeatedly calling function ssrkf45 for
% a single RK Fehlberg 45 step. The truncation error is
% estimated along the solution to adjust the integration
% step according to a specified error tolerance.
%
% Argument list
%
% neqn number of first order ODEs
% t0 initial value of independent variable
% tf final value of independent variable
% u0 initial condition vector of length neqn
% nsteps maximum number of rkf45 steps
% abserr absolute error tolerance
% relerr relative error tolerance
% u ODE solution vector of length neqn after
% nsteps steps
%
% Initial integration step
h=(tf-t0)/2.0;
%
% Minimum allowable step
hmin=(tf-t0)/nsteps;
%
% Start integration
 t=t0;
%
% While independent variable is less than the final
% value, continue the integration
while t <= tf*0.999
%
% If the next step along the solution will go past
% the final value of the independent variable, set the
% step to the remaining distance to the final value
if t+h > tf h=tf-t; end
% Single rkf45 step
[t,u,e]=ssrkf45(neqn,t0,u0,h);

% Flag for the end of the integration
nfin1=1;

% Check if any of the ODEs have violated the error criteria
for i=1:neqn
    if abs(e(i)) > abs(u(i))*relerr+abserr
        % Error violation, so integration is not complete.
        % Reduce integration step because of error violation
        % and repeat integration from the base point
        h=h/2.0; nfin1=0; break;
    end
end

% If the current step is less than the minimum allowable step, set the step to the minimum allowable value and continue integration from new base point
if h < hmin h=hmin; nfin1=1; end

% If there is no error violation, continue the integration from new base point
if nfin1 == 1 u0=u; t0=t;

% Test if integration step can be increased
for i=1:neqn
    if abs(e(i)) > (abs(u(i))*relerr+abserr)/32.0
        % Integration step cannot be increased
        nfin1=0; break;
    end
end

% Increase integration step
if nfin1 == 1 h=h*2.0; end

% Continue for no error violation (nfin1=1)
end
% Continue while
end
%
% End of rkf45b

Program 2.1.13
Integrator rkf45b

rkf45b is basically identical to rkc4b. The only essential difference is the call to ssrkf45 in place of ssrkc4. Also, the test for integration step doubling is

if abs(e(i)) > (abs(u(i))*relerr+abserr)/32.0

i.e., there is a division by 32 rather than by 16 as in ssrkc4 since the RKF45 method is fifth order (rather than fourth order).

rkf45b is called by setting int = 6 in function intpar. The output (listed previously) is essentially the same as from euler2b and rkc4b.

function [t,u,e] = ssrkf45(neqn,t0,u0,h)
%
% Function ssrkf45 computes an ODE solution by the RK
% Fehlberg 45 method for one step along the solution (by
% calls to derv to define the ODE derivative vector). It
% also estimates the truncation error of the solution,
% and applies this estimate as a correction to the solution
% vector.
%
% Argument list
%
% neqn number of first order ODEs
% t0 initial value of independent variable
% u0 initial condition vector of length neqn
% h integration step
% t independent variable
% u ODE solution vector of length neqn after
% one rkf45 step
% e estimate of truncation error of the solu-
%
% Derivative vector at initial (base) point
[ut0]=derv(neqn,t0,u0);

% k1, advance of dependent variable vector and
% independent variable for calculation of k2
k1=h*ut0;
u=u0+0.25*k1;
t=t0+0.25*h;

% Derivative vector at new u, t
[ut]=derv(neqn,t,u);

% k2, advance of dependent variable vector and
% independent variable for calculation of k3
k2=h*ut;
u=u0+(3.0/32.0)*k1...
    +(9.0/32.0)*k2;
t=t0+(3.0/8.0)*h;

% Derivative vector at new u, t
[ut]=derv(neqn,t,u);

% k3, advance of dependent variable vector and
% independent variable for calculation of k4
k3=h*ut;
u=u0+(1932.0/2197.0)*k1...
    -(7200.0/2197.0)*k2...
    +(7296.0/2197.0)*k3;
t=t0+(12.00/13.0)*h;

% Derivative vector at new u, t
[ut]=derv(neqn,t,u);

% k4, advance of dependent variable vector and
% independent variable for calculation of k5
k4=h*ut;
u=u0+( 439.0/ 216.0)*k1...
    -( 8.0       )*k2...
    +(3680.0/ 513.0)*k3...
    -( 845.0/4104.0)*k4;
t=t0+h;

% Derivative vector at new u, t
[ut]=derv(neqn,t,u);

%
% k5, advance of dependent variable vector and
% independent variable for calculation of k6
k5 = h*ut;
u = u0 - (8.0/27.0)*k1...
  + (2.0 )*k2...
  -(3544.0/2565.0)*k3...
  +(1859.0/4104.0)*k4...
  -(11.0/40.0)*k5;
t = t0 + 0.5*h;
%
% Derivative vector at new u, t
[ut] = derv(neqn,t,u);
%
% k6
k6 = h*ut;
%
% Fourth order step
sum4 = u0 + (25.0/216.0)*k1...
  + (1408.0/2565.0)*k3...
  + (2197.0/4104.0)*k4...
  -(1.0/5.0)*k5;
%
% Fifth order step
sum5 = u0 + (16.0/135.0)*k1...
  + (6656.0/12825.0)*k3...
  + (28561.0/56430.0)*k4...
  -(9.0/50.0)*k5...
  +(2.0/55.0)*k6;
t = t0 + h;
%
% Truncation error estimate
e = sum5 - sum4;
%
% Fifth order solution vector (from (4,5) RK pair); % two ways to the same result are listed
% u = sum4 + e;
  u = sum5;
%
% End of ssrkf45

Program 2.1.14
Integrator ssrkf45 for a RKF45 step

The only essential difference for ssrkf45 is the coding of the ODE integration method, i.e.,
• sseuler2—single step modified Euler method ((1, 2) pair)
• ssrk4c—single step classical fourth-order RK method ((2, 4) pair)
• ssrkf45—single step RKF 45 method ((4, 5) pair)

Another possibility would be to write a function analogous to ssrkf45 using the (2, 3) pair discussed previously. The only requirement would be to use the (2, 3) formulas in place of those for the (4, 5) pair; all the other coding in the preceding functions would remain essentially the same. The writing of a (2, 3) integrator is therefore left as an excercise for the reader.

We also have observed in the previous routines an organizing principle that we will use consistently throughout the remainder of this book:

• A set of library routines, which are general purpose and should therefore not be changed from one ODE application to the next. Specifically, the following routines can be applied generally to an ODE problem and should therefore not be modified:

<table>
<thead>
<tr>
<th>Routine Names</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ode1x1</td>
<td>Main program</td>
</tr>
<tr>
<td>euler2a, euler2b, sseuler</td>
<td>(1,2) RK pair</td>
</tr>
<tr>
<td>rkc4a, rkc4b, ssrkc4</td>
<td>(2,4) RK pair</td>
</tr>
<tr>
<td>rkf45a, rkf45b, ssrkf45</td>
<td>(4,5) RK pair</td>
</tr>
</tbody>
</table>

• A set of routines that apply specifically to the ODE problem at hand:

<table>
<thead>
<tr>
<th>Routine Names</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial</td>
<td>Set initial conditions</td>
</tr>
<tr>
<td>dero</td>
<td>Compute ODE RHS vector</td>
</tr>
<tr>
<td>fprintf</td>
<td>Display ODE solution</td>
</tr>
</tbody>
</table>

While these routines pertain to a specific ODE application, in this case the 1x1 ODE problem, they can serve as templates for other ODE applications. In other words, they can be easily modified for the solution of other ODEs. For example, they could be modified from the 1x1 problem of Equations 1.3 and 1.4 to the 2x2 problem of Equations 1.6, 1.16, and 1.17.

• One routine, intpar falls somewhere between the two preceding sets. It is general purpose in the sense that it sets the parameters that control the operation of the general-purpose ODE integration routines; e.g., it sets the error tolerances. On the other hand, the specific values of the parameters depend on the properties of the particular ODE problem; e.g., the number of integration steps, nsteps, may have to be adjusted. Thus, some trial and error generally will be required when coding intpar for a new ODE application. This is particularly the case with the error tolerances as
discussed in Chapter 1; in fact, probably the single most common cause of library integrator failures is the specification of inappropriate error tolerances.

To summarize this organizing principle, the general-purpose (library) routines and the problem-specific routines should be separated as much as possible in the development of new ODE applications. This separation cannot be achieved completely because of the need to tune the integrator parameters, i.e., in \textit{intpar}.

This basically completes the discussion of the ODE integrators in MATLAB. In a subsequent section, we revisit some of the preceding routines to which some output statements are added to indicate what is occurring within them, particularly with regard to error monitoring and step size adjustment (in \textit{euler2b}, \textit{rk4b}, and \textit{rkf45b}). The purpose of this additional output is to (a) observe if the integrators operate internally as we would expect, and (b) present a method for debugging when approaching a new ODE problem. We now proceed to the programming of the preceding integrators in other languages.

### 2.2 Programming in C

Because the operation of the (1, 2), (2, 4), and (4, 5) integrators has been discussed in detail in Section 2.1, we now proceed with the analogous programming in C, but without the same detailed explanation. In other words, we consider the listing of the routines to be essentially self-explanatory (also, we will not consider the details of the differences in syntax between MATLAB and C). Our expectation is that the C code will be largely self-explanatory when compared with the preceding MATLAB code. In fact, we have tried to make the two sets of routines closely resemble each other to facilitate the understanding of one set in terms of the other.

Also, we have used alphabetical ordering of the languages in the following discussion to demonstrate that we do not endorse one language over another. The one exception is the preceding use of MATLAB, which we selected first because of its simplicity in handling vectors (without subscripting). However, this does not mean we favor MATLAB, and in fact, we also have some reservations about this and other “automatic features” possibly leading to unintended errors and consequences. In other words, subscripting within \textit{do} and \textit{for} loops provides an unambiguous indication of how vectors (and matrices) are coded and manipulated with very little additional coding (efficiency may also be a consideration, especially with MATLAB, which tends to be more efficient without subscripting).

Thus, we feel that all of the languages are about comparable in utility for the numerical integration of ODE/PDE systems. A main concern is therefore
to provide code that follows a common theme and format to demonstrate that all of the languages are about equal in terms of their utility in studying differential systems.

To start, we list a main program analogous to Program 2.1.1:

/*
Numerical solution to the 1 x 1 ODE system by six integrators
*/

/* Include headers */
#include <stdio.h>
#include <math.h>
#include "ode1x1.h"
#include "euler.h"
#include "rk.h"

/* Main program */
void main()
{
    /* Type variables */
    double u0[SIZE],u[SIZE];
    double tp;
    int i, j, ncase;

    /* Open a file for output */
    if((fid=fopen("ode1x1c.out","w+"))==NULL)
    {
        printf("Error opening output file\n");
    }

    /* Step through six integrators */
    for(ncase=1;ncase<=6;ncase++)
    {
        /* ODE integration parameters */
        intpar();

        /* Initial condition */
        initial(u0,t0);

        /* Output interval */
        tp=tf-t0;

        /* Step through nout grid points */
        for(j=1;j<=nout;j++)
{ /* Print current solution */
    fprintf(ncase,t0,u0);

    /* Select ODE integrator */
    switch(ncase)
    {
        /* Fixed step modified Euler integrator */
        case 1:
            euler2a(n,t0,tf,u,u0,nsteps);
            break;

        /* Variable step modified Euler integrator */
        case 2:
            euler2b(n,t0,tf,u,u0,nsteps,abserr,relerr);
            break;

        /* Fixed step classical fourth order RK integrator */
        case 3:
            rkc4a(n,t0,tf,u,u0,nsteps);
            break;

        /* Variable step classical fourth order RK integrator */
        case 4:
            rkc4b(n,t0,tf,u,u0,nsteps,abserr,relerr);
            break;

        /* Fixed step RK Fehlberg (RKF45) integrator */
        case 5:
            rkf45a(n,t0,tf,u,u0,nsteps);
            break;

        /* Variable step RK Fehlberg (RKF45) integrator */
        case 6:
            rkf45b(n,t0,tf,u,u0,nsteps,abserr,relerr);
            break;
    }

    /* Advance solution */
    t0=tf;
    tf=tf+tp;
    for(i=1;i<=n;i++)
{ 
    u0[i]=u[i];
}

/* Next output */
}

/* Next integrator */
}

/* Complete solution computed. Close the output file */
close(fid);

/* End of main */
}

Program 2.2.1
C main program for the numerical integration of Equation 1.3 subject to initial condition Equation 1.4

We can note the following points about this program:

• Comments in C are generally enclosed within the delimiters /* */ (which seems safe since the arithmetic operators * and / generally will not be adjacent when writing C code).
• The main program starts with the declaration of some header files:

    /* Include headers */
    #include <stdio.h>
    #include <math.h>
    #include "ode1x1.h"
    #include "euler.h"
    #include "rk.h"

In addition to the standard header files for C (stdio.h and math.h), we have included three header files, ode1x1.h, euler.h and rk.h for the numerical integration of the 1x1 ODE (Equation 1.3) using the (1, 2), (2, 4), and (4, 5) pairs.
• Instead of executing each of the six ODE integrators individually by changing int through the values int = 1, 2, ..., 6, we call all six integrators in a loop:

    /* Step through six integrators */
    for(ncase=1;ncase<=6;ncase++)

Otherwise, the coding in Program 2.2.1 closely parallels that in Program 2.1.1.
Header file *ode1x1.h* is listed below:

```c
/**
 * Definition of functions intpar, inital, derv, fprint
 * for the 1 x 1 ODE system
 */

/* Include headers */
#include <stdio.h>
#include <math.h>

/* Maximum number of ODEs */
#define SIZE 500

/* Type variables as extern (global) */
extern int n, nout, nsteps;
extern double t0, tf, abserr, relerr;
extern FILE *fid;

/* Integration parameters */
void intpar();

/* Initial condition */
void inital(double u0[], double t);

/* Derivative vector */
void derv(double ut[], double t, double u[]);

/* Output */
void fprint(int ncase, double t, double u[]);
```

**Program 2.2.2**

Header file *ode1x1.h* included in Program 2.2.1

All of the routines defined in *ode1x1.h* have already been discussed in detail in Section 2.1. We can note the following details:

- The maximum number of ODES has been set (arbitrarily) at 500:

```c
#define SIZE 500
```

This absolute sizing can be increased to any desired value (although it is more than adequate since we are integrating one ODE).
Several global variables are defined that can be shared between routines:

```c
/* Type variables as extern (global) */
extern int neqn, nout, nsteps;
extern double t0, tf, abserr, relerr;
extern FILE *fid;
```

A file ID (pointer) is also declared as a global variable for writing an external output file.

Routines `intpar`, `inital`, `derv`, and `fprint` are declared. They have the same function as those of the same name in Section 2.1. All these routines are listed subsequently.

Otherwise, the coding in Programs 2.2.1 and 2.2.2 follows directly from their counterparts in Section 2.1.

`intpar` is listed below:

```c
#include "ode1x1.h"

/* Type global variables */

int neqn, nout, nsteps;

double t0, tf, abserr, relerr;

/* Define file ID */
FILE *fid;

void intpar()
{
/* Function intpar sets the parameters to control the integration of the 1 x 1 ODE system */
{
/* Number of ODEs */
    neqn=1;

/* Number of output points */
    nout=6;

/* Maximum number of steps in the interval t0 to tf */
    nsteps=100;

/* Initial, final values of independent variable */
    t0=0.0;
    tf=1.0;
```
/* Error tolerances */
abserr=pow(10,-5);
relerr=pow(10,-5);

/* End of intpar */
}

Program 2.2.3
Routine intpar called in Program 2.2.1 to set the integration parameters

intpar is essentially identical to its counterpart in Program 2.1.3.
initial, derv and fprint are listed below:

void inital(double u0[],double t0)

/* Function inital sets the initial condition vector for the 1 x 1 ODE problem */
{
    /* Initial condition */
    u0[1]=1.0;

    /* End of inital */
}

void derv(double ut[], double t, double u[])

/* Function derv computes the derivative vector of the 1 x 1 ODE problem */
{
    /* Type variables */
    double alpha, lambda;

    /* Problem parameters */
    alpha=1.0;
    lambda=1.0;

    /* Derivative vector */
    ut[1]=lambda*exp(-alpha*t)*u[1];

    /* End of derv */
}

void fprint(int ncase, double t, double u[])
/* Function fprint displays the numerical and exact solutions to the 1 x 1 ODE problem */
{
    /* Type variables */
    double ue[2], diff[2];
    double u0, alpha, lambda;

    /* Problem parameters */
    u0=1.0;
    alpha=1.0;
    lambda=1.0;

    /* Print a heading for the solution at t = 0 */
    if(t<=0.0)
    {
        fprintf(fid,"

        int = %3d
",ncase);

        /* Label for ODE integrator */
        switch(ncase)
        {
            case 1: /* Fixed step modified Euler */
                fprintf(fid,"\n euler2a integrator\n");
                break;

            case 2: /* Variable step modified Euler */
                fprintf(fid,"\n euler2b integrator\n");
                break;

            case 3: /* Fixed step classical fourth order RK */
                fprintf(fid,"\n rkc4a integrator\n");
                break;

            case 4: /* Variable step classical fourth order RK */
                fprintf(fid,"\n rkc4b integrator\n");
                break;

            case 5: /* Fixed step RK Fehlberg 45 */
                fprintf(fid,"\n rkf45a integrator\n");
                break;

            case 6: /* Variable step RK Fehlberg 45 */
                fprintf(fid,"\n rkf45b integrator\n");
                break;
        }
    }
}
/* Heading */
fprintf(fid,\n t u1(num) u1(ex) diff1\n\n);

/* End of t = 0 heading */
}

/* Analytical solution vector */
ue[1]=u0*exp(lambda/alpha*(1.0-exp(-alpha*t)));

/* Difference between exact and numerical solution vectors */
diff[1]=u[1]-ue[1];

/* Display the numerical and exact solutions, and their difference */
fprintf(fid,"%10.2f %10.5f %10.5f %13.4e\n",
 t,u[1],ue[1],diff[1]);

/* End of fprintf */
}

Program 2.2.4
initial, derv and fprintf called in the solution of Equations 1.3 and 1.4

These routines should be self-explanatory, particularly when compared with their MATLAB counterparts in Section 2.1.

The output from the six integrators called in Program 2.2.1 is listed below (written to file ode1x1c.out in fprintf):

int = 1

euler2a integrator

<table>
<thead>
<tr>
<th>t</th>
<th>u1(num)</th>
<th>u1(ex)</th>
<th>diff1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.00000</td>
<td>1.00000</td>
<td>0.0000e+00</td>
</tr>
<tr>
<td>1.00</td>
<td>1.88160</td>
<td>1.88160</td>
<td>-6.4665e-08</td>
</tr>
<tr>
<td>2.00</td>
<td>2.37421</td>
<td>2.37421</td>
<td>3.8793e-06</td>
</tr>
<tr>
<td>3.00</td>
<td>2.58627</td>
<td>2.58626</td>
<td>6.0332e-06</td>
</tr>
<tr>
<td>4.00</td>
<td>2.66895</td>
<td>2.66895</td>
<td>6.9239e-06</td>
</tr>
<tr>
<td>5.00</td>
<td>2.70004</td>
<td>2.70003</td>
<td>7.2649e-06</td>
</tr>
</tbody>
</table>
int =  2

euler2b integrator

<table>
<thead>
<tr>
<th>t</th>
<th>u1(num)</th>
<th>u1(ex)</th>
<th>diff1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.00000</td>
<td>1.00000</td>
<td>0.0000e+00</td>
</tr>
<tr>
<td>1.00</td>
<td>1.88159</td>
<td>1.88160</td>
<td>-6.2196e-06</td>
</tr>
<tr>
<td>2.00</td>
<td>2.37421</td>
<td>2.37421</td>
<td>-3.7783e-06</td>
</tr>
<tr>
<td>3.00</td>
<td>2.58626</td>
<td>2.58626</td>
<td>2.9325e-07</td>
</tr>
<tr>
<td>4.00</td>
<td>2.66895</td>
<td>2.66895</td>
<td>3.6651e-06</td>
</tr>
<tr>
<td>5.00</td>
<td>2.70003</td>
<td>2.70003</td>
<td>6.2494e-06</td>
</tr>
</tbody>
</table>

int =  3

rkc4a integrator

<table>
<thead>
<tr>
<th>t</th>
<th>u1(num)</th>
<th>u1(ex)</th>
<th>diff1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.00000</td>
<td>1.00000</td>
<td>0.0000e+00</td>
</tr>
<tr>
<td>1.00</td>
<td>1.88160</td>
<td>1.88160</td>
<td>-2.7188e-11</td>
</tr>
<tr>
<td>2.00</td>
<td>2.37421</td>
<td>2.37421</td>
<td>-3.2671e-11</td>
</tr>
<tr>
<td>3.00</td>
<td>2.58626</td>
<td>2.58626</td>
<td>-3.4825e-11</td>
</tr>
<tr>
<td>4.00</td>
<td>2.66895</td>
<td>2.66895</td>
<td>-3.5646e-11</td>
</tr>
<tr>
<td>5.00</td>
<td>2.70003</td>
<td>2.70003</td>
<td>-3.5952e-11</td>
</tr>
</tbody>
</table>

int =  4

rkc4b integrator

<table>
<thead>
<tr>
<th>t</th>
<th>u1(num)</th>
<th>u1(ex)</th>
<th>diff1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.00000</td>
<td>1.00000</td>
<td>0.0000e+00</td>
</tr>
<tr>
<td>1.00</td>
<td>1.88160</td>
<td>1.88160</td>
<td>-4.2632e-08</td>
</tr>
<tr>
<td>2.00</td>
<td>2.37421</td>
<td>2.37421</td>
<td>6.8773e-08</td>
</tr>
<tr>
<td>3.00</td>
<td>2.58626</td>
<td>2.58626</td>
<td>9.3031e-08</td>
</tr>
<tr>
<td>4.00</td>
<td>2.66895</td>
<td>2.66895</td>
<td>1.0305e-07</td>
</tr>
<tr>
<td>5.00</td>
<td>2.70003</td>
<td>2.70003</td>
<td>1.4624e-07</td>
</tr>
</tbody>
</table>
int =  5

rKF45a integrator

<table>
<thead>
<tr>
<th>t</th>
<th>u1(num)</th>
<th>u1(ex)</th>
<th>diff1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.00000</td>
<td>1.00000</td>
<td>0.0000e+00</td>
</tr>
<tr>
<td>1.00</td>
<td>1.88160</td>
<td>1.88160</td>
<td>6.6613e-16</td>
</tr>
<tr>
<td>2.00</td>
<td>2.37421</td>
<td>2.37421</td>
<td>2.6645e-15</td>
</tr>
<tr>
<td>3.00</td>
<td>2.58626</td>
<td>2.58626</td>
<td>1.3323e-15</td>
</tr>
<tr>
<td>4.00</td>
<td>2.66895</td>
<td>2.66895</td>
<td>1.3323e-15</td>
</tr>
<tr>
<td>5.00</td>
<td>2.70003</td>
<td>2.70003</td>
<td>-1.3323e-15</td>
</tr>
</tbody>
</table>

int =  6

rKF45b integrator

<table>
<thead>
<tr>
<th>t</th>
<th>u1(num)</th>
<th>u1(ex)</th>
<th>diff1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.00000</td>
<td>1.00000</td>
<td>0.0000e+00</td>
</tr>
<tr>
<td>1.00</td>
<td>1.88159</td>
<td>1.88160</td>
<td>-3.2870e-06</td>
</tr>
<tr>
<td>2.00</td>
<td>2.37421</td>
<td>2.37421</td>
<td>-4.6026e-06</td>
</tr>
<tr>
<td>3.00</td>
<td>2.58626</td>
<td>2.58626</td>
<td>-5.0371e-06</td>
</tr>
<tr>
<td>4.00</td>
<td>2.66894</td>
<td>2.66895</td>
<td>-5.1682e-06</td>
</tr>
<tr>
<td>5.00</td>
<td>2.70002</td>
<td>2.70003</td>
<td>-5.2117e-06</td>
</tr>
</tbody>
</table>

We can note the following details about this output:

- The error tolerances set in intpar
  
  /* Error tolerances */
  abserr=pow(10,-5);
  relerr=pow(10,-5);

  are satisfied by all of the integrators.

- The fixed step integrators, euler2a, rkc4a, and rkf45a exceed the error tolerances by orders of magnitude when using 100 integration steps in each output interval as set in intpar:

  /* Maximum number of steps in the interval t0 to tf */
  nsteps=100;

  In fact, rkc4a and rkf45a provide accuracies that might be considered excessive. In the case of rkf45a, the errors are of the order of the machine epsilon (unit roundoff, machine precision) for C.
We now consider the six integrators by listing and briefly discussing the source code (again, we are relying on the preceding discussion of the six integrators in MATLAB to explain most of the details). *euler2a* is listed first:

```c
#include "euler.h"
#include <stdio.h>

double euler2a(int neqn, double t0, double tf, double u[], double u0[], int nsteps)
{
    double e[SIZE];
    double h,t;
    int i,j;

    h=(tf-t0)/nsteps;

    for(i=1;i<=nsteps;i++)
    {
        t=sseuler(neqn,t0,u0[],h,u,e);
    }

    return t;
}
```

Function *euler2a* computes an ODE solution by the fixed step modified Euler method for a series of points along the solution by repeatedly calling function *sseuler* for a single modified Euler step.

**Argument list**

- `neqn` number of first order ODEs
- `t0` initial value of independent variable
- `tf` final value of independent variable
- `u0` initial condition vector of length `neqn`
- `nsteps` number of modified Euler steps
- `uODE` solution vector of length `neqn` after `nsteps` steps

* */
/ * Reset base point values for next modified Euler step */
for(j=1;j<=neqn;j++)
{
    u0[j]=u[j];
}
t0=t;

/* End for */
}

return 0;

/* End of euler2a */
}

Program 2.2.5
Integrator euler2a

euler2a closely parallels the MATLAB routine of Program 2.1.5 and therefore will not be discussed further.
euler2b is listed below:

double euler2b(int neqn, double t0, double tf, double u[],
    double u0[], int nsteps, double abserr,
    double relerr)

/*
Function euler2b computes an ODE solution by the variable step modified Euler method for a series of points along the solution by repeatedly calling function ss euler for a single modified Euler step. The truncation error is estimated along the solution to adjust the integration step according to a specified error tolerance.

Argument list

    neqn    number of first order ODEs
    t0      initial value of independent variable
    tf      final value of independent variable
    u0      initial condition vector of length neqn
    nsteps  number of modified Euler steps
*/
u ODE solution vector of length neqn after nsteps steps

/*
 /* Type variables */
 double h, hmin, t;
 int i, nfin1;
 double e[SIZE];

 /* Integration step */
 h=(tf-t0)/8.0;

 /* Minimum allowable step */
 hmin=(tf-t0)/(float)(nsteps);

 /* Start integration */
 t=t0;

 /* While independent variable is less than the final value, continue the integration */
 while (t <= tf*0.999)
 {
  /* If the next step along the solution will go past the final value of the independent variable, set the step to the remaining distance to the final value */
  if((t+h) > tf)
  {
   h=tf-t;
  }
  /* Single modified Euler step */
  t=ss euler(neqn,t0,u0,h,u,e);

  /* Flag for the end of the integration */
  nfin1=1;

  /* Check if any of the ODEs have violated the error */
  for(i=1;i<=neqn;i++)
  {
   if(fabs(e[i]) > (fabs(u[i])*relerr+abserr))
   {
    /* Error violation, so integration is incomplete. Reduce integration step because of error violation and repeat integration from base point */
   }
nfin1=0;
h=h/2.0;

/* If the current step is less than the minimum allowable step, set the step to the minimum allowable value and continue integration from new base point */
if(h < hmin)
{
    h=(float)hmin;
nfin1=1;
}
break;
}

/* If there is no error violation, continue the integration from the new base point */
if(nfin1 == 1 )
{
    for(i=1;i<=neqn;i++)
    {
        u0[i]=u[i];
    }
t0=t;

    /* Test if integration step can be increased */
    for(i=1;i<=neqn;i++)
    {
        if(fabs(e[i]) > ((fabs(u[i])*relerr +abserr)/4.0))
        {
            /* Integration step cannot be increased */
            nfin1=0;
            break;
        }
    }

    /* Increase integration step */
    if(nfin1 == 1)
    {
        h=h*2.0;
    }

    /* End if */
}
Program 2.2.6
Integrator euler2b

Again, euler2b closely parallels the MATLAB routine of Program 2.1.8 except for differences in syntax between MATLAB and C and, therefore, will not be discussed further.

sseuler is listed below:

double sseuler(int neqn, double t0, double u0[], double h, double u[], double e[])

/*
Function sseuler computes an ODE solution by the modified Euler method for one step along the solution (by calls to derv to define the ODE derivative vector). It also estimates the truncation error of the solution, and applies this estimate as a correction to the solution vector.

Argument list

neqn number of first order ODEs

t0 initial value of independent variable

u0 initial condition vector of length neqn

h integration step

t independent variable

u ODE solution vector of length neqn after one modified Euler step

e estimate of truncation error of the solution vector
*/
{ /* Type variables */
    double ut0[SIZE], ut[SIZE];
    double t;
    int j;

    /* Derivative vector at initial (base) point */
    derv(ut0,t0,u0);

    /* First order (Euler) step */
    for(j=1;j<=neqn;j++)
    {
        u[j]=u0[j]+ut0[j]*h;
    }
    t=t0+h;

    /* Derivative at advance point */
    derv(ut,t,u);

    /* Second order (modified Euler) step */
    for(j=1;j<=neqn;j++)
    {
        /* Truncation error estimate */
        e[j]=(ut[j]-ut0[j])*h/2.0;
        /* Second order (modified Euler) solution vector */
        u[j]=u[j]+e[j];
    }

    return t;

    /* End of sseuler */
}

Program 2.2.7
Integrator sseuler for a single modified Euler step

Again, sseuler is a direct analog of the MATLAB routine in Program 2.1.6. The header file for euler2a, euler2b, and sseuler is as follows:

/*

Routines for the Euler ODE Integration

*/
#include <math.h>
#define SIZE 500

double euler2a(int neqn, double t0, double tf, double u[],
                double u0[], int nsteps);

double euler2b(int neqn, double t0, double tf, double u[],
                double u0[], int nsteps, double abserr,
                double relerr);

double sseuler(int neqn, double t0, double u0[], double h,
                double u[], double e[]);

Program 2.2.8
Header file euler.h for euler2a, euler2b, and sseuler

rkc4a, rkc4b, and ssrkc4 are listed below without further comment (their
MATLAB counterparts are Programs 2.1.9, 2.1.10, and 2.1.11):

#include "rk.h"

double rkc4a(int neqn, double t0, double tf, double u[],
            double u0[], int nsteps)
/*
 Function rkc4a computes an ODE solution by the fixed step
 classical fourth order RK method for a series of points
 along the solution by repeatedly calling function ssrkc4
 for a single classical fourth order RK step.

 Argument list

 neqn  number of first order ODEs
 t0    initial value of independent variable
 tf    final value of independent variable
 u0    initial condition vector of length neqn
 nsteps number of rkc4 steps
 u     ODE solution vector of length neqn after
       nsteps steps
*/


*/
{
  /* Type variables */
  int i, j;
  double t, h, e[SIZE];

  /* Integration step */
  h = (tf-t0)/nsteps;

  /* nsteps rkc4 steps */
  for (i=1; i<=nsteps; i++)
  {
    /* Single rkc4 step */
    t = ssrkc4(neqn,t0,u0,h,u,e);

    /* Reset base point values for next rk4c step */
    for (j=1; j<=neqn; j++)
    {
      u0[j] = u[j];
    }
    t0 = t;
  }

  return 0;

  /* End of rkc4a */
}

Program 2.2.9
Integrator rkc4a

double rkc4b(int neqn, double t0, double tf, double u[],
              double u0[], int nsteps,double abserr,
              double relerr)

  /*
   * Function rkc4b computes an ODE solution by a variable
   * step classical fourth order RK method for a series of
   * points along the solution by repeatedly calling function
   * ssrkc4 for a single classical fourth order RK step. The
   * truncation error is estimated along the solution to
   * adjust the integration step according to a specified
   * error tolerance.
   */
Argument list

neqn number of first order ODEs
t0 initial value of independent variable
tf final value of independent variable
u0 initial condition vector of length neqn
nsteps maximum number of rkc4 steps
abserr absolute error tolerance
relerr relative error tolerance
u ODE solution vector of length neqn after nsteps steps

/* Type variables */
double h, hmin, t, e[SIZE];
int nfin1, i;

/* Integration step */
h=(tf-t0)/2.0;

/* Minimum allowable step */
hmin=(tf-t0)/nsteps;

/* Start integration */
t=t0;

/* While independent variable is less than the final value, continue the integration */
while(t <= (tf*0.999))
{
   /* If the next step along the solution will go past the final value of the independent variable, set the step to the remaining distance to the final value */
   if((t+h) > tf)
   {
      h=tf-t;
   }
/* Single rkc4 step */
t=ssrkc4(neqn,t0,u0,h,u,e);

/* Flag for the end of the integration */
nfin1=1;

/* Check if any of the ODEs have violated the error criterion */
for(i=1;i<=neqn;i++)
{
    if( fabs(e[i]) > (fabs(u[i])*relerr+abserr))
    {
        /* Error violation, so integration is incomplete. Reduce integration step because of error violation and repeat integration from base point */
        nfin1=0;
        h=h/2.0;

        /* If the current step is less than the minimum allowable step, set the step to the minimum allowable value and continue integration from new base point */
        if(h < hmin)
        {
            h = hmin;
            nfin1=1;
        }
        break;
    }
}

/* If there is no error violation, continue the integration from the new base point */
if(nfin1 == 1)
{
    for(i=1;i<=neqn;i++)
    {
        u0[i]=u[i];
    }
    t0=t;

    /* Test if integration step can be increased */
    for(i=1;i<=neqn;i++)
if(fabs(e[i]) > ((fabs(u[i])*relerr + abserr)/16.0))
{
    /* Integration step cannot be increased */
    nfin1=0;
    break;
}

/* Increase integration step */
if(nfin1 == 1)
{
    h=h*2.0;
}

/* End if */

/* End while */

return 0;

/* End of rkc4b */
}

Program 2.2.10
Integrator rkc4b

double ssrkc4(int neqn, double t0, double u0[], double h, double u[], double e[])
/
    Function ssrkc4 computes an ODE solution by the classical fourth order RK method for one step along the solution (by calls to derv to define the ODE derivative vector). It also estimates the truncation error of the solution, and applies this estimate as a correction to the solution vector.

Argument list

    neqn number of first order ODEs
    t0 initial value of independent variable
u0 initial condition vector of length neqn
h integration step
t independent variable
u ODE solution vector of length neqn after one rkc4 step
e estimate of truncation error of the solution vector

/*
*/
{
  /* Type variables */
double ut0[SIZE], ut[SIZE], u4[SIZE];
double t, k1[SIZE], k2[SIZE], k3[SIZE], k4[SIZE];
int j;

  /* Derivative vector at initial (base) point */
derv(ut0,t0,u0);

  /* k1; stepping for k2 */
  for(j=1;j<=neqn;j++)
  {
    k1[j]=h*ut0[j];
    u[j]=u0[j]+0.5*k1[j];
  }
  t=t0+0.5*h;

  /* Derivative vector at next RK point */
derv(ut,t,u);

  /* k2; stepping for k3 */
  for(j=1;j<=neqn;j++)
  {
    k2[j]=h*ut[j];
    u[j]=u0[j]+0.5*k2[j];
  }
  t=t0+0.5*h;

  /* Derivative vector at next RK point */
derv(ut,t,u);

  /* k3; stepping for k4 */
  for(j=1;j<=neqn;j++)
  {
{
    k3[j]=h*ut[j];
    u[j]=u0[j]+k3[j];
}
t=t0+h;
/* Derivative vector at next RK point */
derv(ut,t,u);

/* k4; second and fourth order step; error estimate; error correction */
for(j=1;j<=neqn;j++)
{
    k4[j]=h*ut[j];
    u[j]=u0[j]+k2[j];
    u4[j]=u0[j]+(1.0/6.0)*(k1[j]+2.0*k2[j]+2.0*k3[j]+
        k4[j]);
    e[j]=u4[j]-u[j];
    u[j]=u[j]+e[j];
}
t=t0+h;
return t;
/* End of ssrk4c */
}

Program 2.2.11
Integrator ssrk4 for a classical fourth-order RK step

rkf45a, rkf45b, and ssrkf45 are listed below without further comment (their
MATLAB counterparts are Programs 2.1.9, 2.1.10, and 2.1.11):

double rkf45a(int neqn, double t0, double tf, double u[],
    double u0[], int nsteps)
/*
   Function rkf45a computes an ODE solution by the fixed
   step RK Fehlberg 45 RK method for a series of points
   along the solution by repeatedly calling function ssrkf45
   for a single RK Fehlberg 45 step.

Argument list

    neqn       number of first order ODEs

    t0         initial value of independent variable

tf final value of independent variable
u0 initial condition vector of length neqn
nsteps number of rkf45 steps
u ODE solution vector of length neqn after nsteps steps
*/
{
  /* Type variables */
  double t, h, e[SIZE];
  int i,j;

  /* Integration step */
  h=(tf-t0)/nsteps;

  /* nsteps rkf45 steps */
  for(i=1;i<=nsteps;i++)
  {
    /* Single rkf45 step */
    t=ssrkf45(neqn,t0,u0,h,u,e);

    /* Reset base point values for next rkf45 step */
    for(j=1;j<=neqn;j++)
      {u0[j]=u[j];}
    t0=t;
  }

  return 0;

  /* End rkf45a */
}

Program 2.2.12
Integrator rkf45a
double rkf45b(int neqn, double t0, double tf, double u[],
               double u0[], int nsteps, double abserr,
               double relerr)
  /*
  Function rkf45b computes an ODE solution by a variable
  step classical RK Fehlberg 45 method for a series of
points along the solution by repeatedly calling function sssrkf45 for a single RK Fehlberg 45 step. The truncation error is estimated along the solution to adjust the integration step according to a specified error tolerance.

Argument list

- neqn: number of first order ODEs
- t0: initial value of independent variable
- tf: final value of independent variable
- u0: initial condition vector of length neqn
- nsteps: maximum number of rkf45 steps
- abserr: absolute error tolerance
- relerr: relative error tolerance
- u: ODE solution vector of length neqn after nsteps steps

*/
{
    /* Type variables */
    double h, hmin, t, e[SIZE];
    int i, nfin1;

    /* Integration step */
    h=(tf-t0)/2.0;

    /* Minimum allowable step */
    hmin=(tf-t0)/nsteps;

    /* Start integration */
    t=t0;

    /* While independent variable is less than the final value, continue the integration */
    while(t <= (tf*0.999))
    {
    /* End of code */
}
/* If the next step along the solution will go past
the final value of the independent variable, set the
step to the remaining distance to the final value */
if((t+h) > tf)
{
    h=tf-t;
}

/* Single rkf45 step */
t=ssrkf45(neqn,t0,u0,h,u,e);

/* Flag for the end of the integration */
nfin1=1;

/* Check if any of the ODEs have violated the error
criterion */
for (i=1;i<=neqn;i++)
{
    if(fabs(e[i]) > (fabs(u[i])*relerr+abserr))
    {
        /* Error violation, so integration is
           incomplete. Reduce integration step because
           of error violation and repeat integration
           from base point */
        nfin1=0;
        h=h/2.0;
        /* If the current step is less than the
           minimum allowable step, set the step to the
           minimum allowable value and continue
           integration from new base point */
        if(h < hmin)
        {
            h = hmin;
            nfin1=1;
        }
        break;
    }
}

/* If there is no error violation, continue the
integration from the new base point */
if(nfin1 == 1)
{
double ssrkf45(int neqn, double t0, double u0[], double h, double u[], double e[])
/*
 Function ssrkf45 computes an ODE solution by the RK
 Fehlberg 45 method for one step along the solution (by
calls to derv to define the ODE derivative vector). It also estimates the truncation error of the solution, and applies this estimate as a correction to the solution vector.

Argument list

neqn number of first order ODEs

t0 initial value of independent variable

u0 initial condition vector of length neqn

h integration step

t independent variable

u ODE solution vector of length neqn after one rkf45 step

e estimate of truncation error of the solution vector

*/
{
 /* Type variables */
 double t, ut0[SIZE], ut[SIZE], u5[SIZE];
 double k1[SIZE], k2[SIZE], k3[SIZE], k4[SIZE], k5[SIZE],
 k6[SIZE];
 int j;

 /* Derivative vector at initial (base) point */
derv(ut0,t0,u0);

 /* k1; stepping for k2 */
 for(j=1;j<=neqn;j++)
 {
   k1[j]=h*ut0[j];
   u[j]=u0[j]+0.25*k1[j];
 }
 t=t0+0.25*h;

 /* Derivative vector at next RK point */
derv(ut,t,u);
/* k2; stepping for k3 */
for(j=1;j<=neqn;j++)
  {
    k2[j]=h*ut[j];
    u[j]=u0[j]+(3.0/32.0)*k1[j]
         +(9.0/32.0)*k2[j];
  }
t=t0+(3.0/8.0)*h;

/* Derivative vector at next RK point */
derv(ut,t,u);

/* k3; stepping for k4 */
for(j=1;j<=neqn;j++)
  {
    k3[j]=h*ut[j];
    u[j]=u0[j]+(1932.0/2197.0)*k1[j]  
             -(7200.0/2197.0)*k2[j]  
             +(7296.0/2197.0)*k3[j];
  }
t=t0+(12.00/13.0)*h;

/* Derivative vector at next RK point */
derv(ut,t,u);

/* k4; stepping for k5 */
for(j=1;j<=neqn;j++)
  {
    k4[j]=h*ut[j];
    u[j]=u0[j]+(439.0/216.0)*k1[j]  
             -(8.0)*k2[j]  
             +(3680.0/513.0)*k3[j]  
             +(845.0/1040.0)*k4[j];
  }
t=t0+h;

/* Derivative vector at next RK point */
derv(ut,t,u);

/* k5; stepping for k6 */
for(j=1;j<=neqn;j++)
  {
    k5[j]=h*ut[j];
\[ u[j] = u0[j] - \left(\frac{8.0}{27.0}\right) k1[j] + 2.0 k2[j] - \left(\frac{3544.0}{2565.0}\right) k3[j] + \left(\frac{1859.0}{4104.0}\right) k4[j] - \left(\frac{11.0}{40.0}\right) k5[j]; \]

\[ t = t_0 + 0.5h; \]

/* Derivative vector at next RK point */
derv(ut, t, u);

/* k6; fourth and fifth order step; error estimate; error correction */
for (j=1; j<=neqn; j++)
{
    k6[j] = h*ut[j];
    u[j] = u0[j] + \left(\frac{25.0}{216.0}\right) k1[j] + \left(\frac{1408.0}{2565.0}\right) k3[j] + \left(\frac{2197.0}{4104.0}\right) k4[j] - \left(\frac{1.0}{5.0}\right) k5[j];
    u5[j] = u0[j] + \left(\frac{16.0}{135.0}\right) k1[j] + \left(\frac{6656.0}{12825.0}\right) k3[j] + \left(\frac{28561.0}{56430.0}\right) k4[j] - \left(\frac{9.0}{50.0}\right) k5[j] + \left(\frac{2.0}{55.0}\right) k6[j];
    e[j] = u5[j] - u[j];
    u[j] = u[j] + e[j];
}

t = t0 + h;

return t;

/* End of ssrkf45 */
}

**Program 2.2.14**
Integrator ssrkf45 for a RKF45 step

The header file for rkc4a, rkc4b, ssrkfc4, rkf45a, rkf45b, and ssrkf45 is as follows:

/*

Routines for the RK Integration

*/
#include <math.h>

#define SIZE 500

double rkc4a(int neqn, double t0, double tf, double u[], double u0[], int nsteps);

double rkc4b(int neqn, double t0, double tf, double u[], double u0[], int nsteps, double abserr, double relerr);

double ssrkc4(int neqn, double t0, double u0[], double h, double u[], double e[]);

double rkf45a(int neqn, double t0, double tf, double u[], double u0[], int nsteps);

double rkf45b(int neqn, double t0, double tf, double u[], double u0[], int nsteps, double abserr, double relerr);

double ssrkf45(int neqn, double t0, double u0[], double h, double u[], double e[]);

Program 2.2.15
Header file for rkc4a, rkc4b, ssrkc4, rkf45a, rkf45b, ssrkf45

This completes the discussion of the C solution of the 1x1 ODE problem of Equations 1.3 to 1.5. We now proceed to the programming of this problem in C++.

2.3 Programming in C++

We present the programming for the remaining languages according to the organizational principle discussed previously; i.e., the general-purpose routines are listed first, followed by the application-specific routines, and then possibly any routines that are somewhere in between. To conclude the discussion of a specific language, we then list the output from the entire group of routines.

Thus, we start with the listing of the main program and ODE integrator routines:
Numerical solution to the 1 x 1 ODE system by six integrators

*/

#include <math.h>
#include <stdlib.h>
#include <iostream.h>
#include <fstream.h>
#include "ode1x1.h"
#include "Euler.h"
#include "RK.h"

/* Main program */
void main()
{
    /* Type variables */
    double tp;
    int i,j,ncase;

    /* Derive objects */
    DEF o1;
    Euler e1;
    RK rk;

    /* Open a file for output */
    ofstream fout("ode1x1cpp.out",ios::out);

    /* Step through six integrators */
    for(ncase=1; ncase<=6; ncase++)
    {
        /* ODE integration parameters */
        o1.intpar();

        /* Initial condition */
        o1.initial();

        /* Output interval */
        tp=o1.tf-o1.t0;
/ * Step through nout grid points */
for(j=1;j<=o1.nout;j++)
{
    /* Print solution */
    o1.fprint(fout,ncase,o1.neqn,o1.t0,o1.u0);

    /* Select ODE integrator */
    switch(ncase)
    {
        /* Fixed step modified Euler integrator */
        case 1:
            e1.euler2a(o1.neqn,o1.t0,o1.tf,o1.u,o1.u0,
                       o1.nsteps);
            break;

        /* Variable step modified Euler integrator */
        case 2:
            e1.euler2b(o1.neqn,o1.t0,o1.tf,o1.u,o1.u0,
                       o1.nsteps,o1.abserr,o1.relerr);
            break;

        /* Fixed step classical fourth order RK integrator */
        case 3:
            rk.rkc4a(o1.neqn,o1.t0,o1.tf,o1.u,o1.u0,
                     o1.nsteps);
            break;

        /* Variable step classical fourth order RK integrator */
        case 4:
            rk.rkc4b(o1.neqn,o1.t0,o1.tf,o1.u,o1.u0,
                     o1.nsteps,o1.abserr,o1.relerr);
            break;

        /* Fixed step RK Fehlberg (RKF45) integrator */
        case 5:
            rk.rkf45a(o1.neqn,o1.t0,o1.tf,o1.u,o1.u0,
                      o1.nsteps);
            break;

        /* Variable step RK Fehlberg (RKF45) integrator */
    }
}
case 6:
    rk.rkf45b(o1.neqn,o1.t0,o1.tf,o1.u,o1.u0,
        o1.nsteps,o1.abserr,o1.relerr);
    break;

/* Advance solution */
    o1.t0=o1.tf;
    o1.tf+=tp;
    for(i=1;i<=o1.neqn;i++)
    {
        o1.u0[i]=o1.u[i];
    }

    /* Next output */

/* Next integrator */

/* Complete solution computed. Close output file */
    fout<<endl;
    fout.close();

/* End of main */

Program 2.3.1
C++ main program for the numerical integration of Equation 1.3 subject to initial condition Equation 1.4

Note, again, as in Program 2.2.1, that all six integrators are called within a single loop.

/* Step through six integrators */
    for(ncase=1;ncase<=6;ncase++)

Also, an output file, _ode1x1cpp.out_ is written by this main program. The associated header files are as follows:

/*

Definition of functions intpar, initial, derv, fprint for the 1 x 1 ODE system

*/
/*

/* Include headers */
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <fstream.h>
#include <iomanip.h>
#include "MOL.h"

/* Type variables as extern (global) */
extern FILE *fid;

class DEF::public MOL
 {
  public:

    /* Integration parameters */
    void intpar();

    /* Initial condition */
    void inital();

    /* Derivative vector */
    void derv(double ut[],double t,double u[]);

    /* Output */
    void fprint(int ncase, int neqn, double t, double u[]);

    void fprint(ofstream &fout, int ncase, int neqn,
               double t, double u[]);
  }

/*

Define the common (global) variables for the
1 x 1 ODE problem

*/

/* Maximum (default) number of ODEs */
#define SIZE 500
class MOL
{
 public:

  
/* Variables for ODE integration */

  int neqn, nout, nsteps;
  double t0, tf, abserr, relerr;
  double u[SIZE], u0[SIZE], e[SIZE];
};

Program 2.3.2
Header files ode1x1.h and MOL.h used by Program 2.3.1

The ODE integration routines are listed below (ode1x1.h is copied to DEF.h before execution):

#include "DEF.h"
#include "Euler.h"

double Euler::euler2a(int neqn, double t0, double tf, 
double u[], double u0[], int nsteps)
  
/*
 Function euler2a computes an ODE solution by the fixed
 step modified Euler method for a series of points along
 the solution by repeatedly calling function sseuler for
 a single modified Euler step.

 Argument list

   neqn     number of first order ODEs
   t0       initial value of independent variable
   tf       final value of independent variable
   u0       initial condition vector of length neqn
   nsteps   number of modified Euler steps
   u        ODE solution vector of length neqn after
             nsteps steps
*/
/*

/* Type variables */
double e[SIZE];
double h, t;
int i, j;

/* Integration step */
h = (tf - t0) / nsteps;

/* nsteps modified Euler steps */
for (i = 1; i <= nsteps; i++)
{
    /* Single modified Euler step */
    t = sseuler(neqn, t0, u0, h, u, e);

    /* Reset base point values for next modified Euler step */
    for (j = 1; j <= neqn; j++)
    {
        u0[j] = u[j];
    }
    t0 = t;

    /* End for */
}

return 0;

/* End of euler2a */
}

Program 2.3.3
Integrator euler2a

double Euler::euler2b(int neqn, double t0, double tf,
    double u[], double u0[], int nsteps, double abserr, double relerr)
/*
    Function euler2b computes an ODE solution by the variable step modified Euler method for a series of points along the solution by repeatedly calling function sseuler for a single modified Euler step. The truncation error is
estimated along the solution to adjust the integration
step according to a specified error tolerance.

Argument list

neqn  number of first order ODEs

 t0  initial value of independent variable

 tf  final value of independent variable

u0  initial condition vector of length neqn

nsteps  number of modified Euler steps

u  ODE solution vector of length neqn after
nsteps steps

/*
 */
{ /* Type variables */
  double h, hmin,t;
  int i, nfin1;
  double e[SIZE];

  /* Integration step */
  h=(tf-t0)/8.0;

  /* Minimum allowable step */
  hmin=(tf-t0)/(float)(nsteps);

  /* Start integration */
  t=t0;

  /* While independent variable is less than the final */
  while(t <= tf*0.999)
  {
    /* If the next step along the solution will go past
the final value of the independent variable, set the
step to the remaining distance to the final value */
    if((t+h) > tf)
    {
      h=tf-t;
    }
    /* Single modified Euler step */
    t=sseuler(neqn,t0,u0,h,u,e);
/* Flag for the end of the integration */
nfin1=1;

/* Check if any of the ODEs have violated the error criterion */
for(i=1;i<=neqn;i++)
{
    if(fabs(e[i]) > (fabs(u[i])*relerr+abserr))
    {
        /* Error violation, so integration is incomplete. Reduce integration step because of error violation and repeat integration from base point */
        nfin1=0;
        h=h/2.0;

        /* If the current step is less than the minimum allowable step, set the step to the minimum allowable value and continue integration from new base point */
        if(h< hmin)
        {
            h = (float)hmin;
            nfin1=1;
        }
        break;
    }
}

/* If there is no error violation, continue the integration from the new base point */
if(nfin1 == 1)
{
    for(i=1;i<=neqn;i++)
    {
        u0[i]=u[i];
    }
    t0=t;

    /* Test if integration step can be increased */
    for(i=1;i<=neqn;i++)
    {
        if(fabs(e[i]) > ((fabs(u[i])*relerr+abserr)/4.0))
        {
            /* Integration step cannot be increased */
        }
    }
}
nfin1=0;
break;
}

/* Increase integration step */
if(nfin1 == 1)
{
    h=h*2.0;
}
/* End if */

/* End while */
return 0;
/* End of euler2b */
}

Program 2.3.4
Integrator euler2b

double Euler::sseuler(int neqn, double t0, double u0[],
                        double h, double u[], double e[])
  /*
    Function sseuler computes an ODE solution by the modified
    Euler method for one step along the solution (by calls to
derv to define the ODE derivative vector). It also
    estimates the truncation error of the solution, and
    applies this estimate as a correction to the solution
    vector.
    
    Argument list
    
    neqn number of first order ODEs
    t0 initial value of independent variable
    u0 initial condition vector of length neqn
    h integration step
  */

t independent variable

u ODE solution vector of length neqn after one modified Euler step

e estimate of truncation error of the solution vector

*/
{
    /* Type variables */
    double ut0[SIZE], ut[SIZE];
    double t;
    int j;

    /* Derivative vector at initial (base) point */
    derv(ut0, t0, u0);

    /* First order (Euler) step */
    for(j=1;j<=neqn;j++)
    {
        u[j]=u0[j]+ut0[j]*h;
    }
    t=t0+h;

    /* Derivative at advance point */
    derv(ut, t, u);

    /* Second order (modified Euler) step */
    for(j=1;j<=neqn;j++)
    {
        /* Truncation error estimate */
        e[j]=(ut[j]-ut0[j])*h/2.0;

        /* Second order (modified Euler) solution vector */
        u[j]=u[j]+e[j];
    }

    return t;

    /* End of sseuler */
}

Program 2.3.5
Integrator sseuler for a single modified Euler step
Routines for the Euler ODE Integration

class Euler:public DEF
{
    public:
    double euler2a(int neqn, double t0, double tf, double u[],
                     double u0[], int nsteps);
    double euler2b(int neqn, double t0, double tf, double u[],
                     double u0[], int nsteps, double abserr,
                     double relerr);
    double sseuler(int neqn, double t0, double u0[], double h,
                    double u[], double e[]);
};

Program 2.3.6
Header file euler.h for euler2a, euler2b, and sseuler

#include "DEF.h"
#include "RK.h"

double RK::rkc4a(int neqn, double t0, double tf,
                 double u[], double u0[], int nsteps)
/*
  Function rkc4a computes an ODE solution by the fixed step
  classical fourth order RK method for a series of points
  along the solution by repeatedly calling function ssrkc4
  for a single classical fourth order RK step.

Argument list
  neqn  number of first order ODEs
  t0    initial value of independent variable
  tf    final value of independent variable
u0 initial condition vector of length neqn
nsteps number of rkc4 steps
u ODE solution vector of length neqn after nsteps steps
*/
{
    /* Type variables */
    int i, j;
    double t, h, e[SIZE];

    /* Integration step */
    h=(tf-t0)/nsteps;

    /* nsteps rkc4 steps */
    for(i=1; i<=nsteps; i++)
    {
        /* Single rkc4 step */
        t=ssrkc4(neqn,t0,u0,h,u,e);

        /* Reset base point values for next rk4c step */
        for(j=1; j<=neqn; j++)
        {
            u0[j]=u[j];
        }
        t0=t;
    }

    return 0;

    /* End of rkc4a */
}

Program 2.3.7
Integrator rkc4a

double RK::rkc4b(int neqn, double t0, double tf,
                 double u[], double u0[], int nsteps,
                 double abserr, double relerr)
/*
 Function rkc4b computes an ODE solution by a variable step classical fourth order RK method for a series of points along the solution by repeatedly calling function
ssrkc4 for a single classical fourth order RK step. The truncation error is estimated along the solution to adjust the integration step according to a specified error tolerance.

Argument list

neqn number of first order ODEs
t0 initial value of independent variable
tf final value of independent variable
u0 initial condition vector of length neqn
nsteps maximum number of rkc4 steps
abserr absolute error tolerance
relerr relative error tolerance
u ODE solution vector of length neqn after nsteps steps

/*
 */
{
    /* Type variables */
    double h, hmin, t, e[SIZE];
    int nfin1, i;

    /* Integration step */
    h=(tf-t0)/2.0;

    /* Minimum allowable step */
    hmin=(tf-t0)/nsteps;

    /* Start integration */
    t=t0;

    /* While independent variable is less than the final value, continue the integration */
    while(t <= (tf*0.999))
    {
        /* If the next step along the solution will go past the final value of the independent variable, set the step to the remaining distance to the final value */
if((t+h) > tf)
{
    h=tf-t;
}

/* Single rkc4 step */
t=ssrkc4(neqn,t0,u0,h,u,e);

/* Flag for the end of the integration */
nfin1=1;

/* Check if any of the ODEs have violated the error criterion */
for(i=1;i<=neqn;i++)
{
    if(fabs(e[i]) > (fabs(u[i])*relerr+abserr))
    {
        /* Error violation, so integration is incomplete. Reduce integration step because of error violation and repeat integration from base point */
        nfin1=0;
        h=h/2.0;

        /* If the current step is less than the minimum allowable step, set the step to the minimum allowable value and continue integration from new base point */
        if(h < hmin)
        {
            h=hmin;
            nfin1=1;
        }
        break;
    }
}

/* If there is no error violation, continue the integration from the new base point */
if(nfin1 == 1)
{
    for(i=1;i<=neqn;i++)
    {
        u0[i]=u[i];
    }
    t0=t;
/* Test if integration step can be increased */
for(i=1;i<=neqn;i++)
{
  if(fabs(e[i]) > ((fabs(u[i])*relerr
       +abserr)/16.0))
  {
       /* Integration step cannot be increased */
       nfin1=0;
       break;
  }
}

/* Increase integration step */
if(nfin1 == 1)
{
  h=h*2.0;
}
/* End if */
/* End while */
}
return 0;
/* End of rkc4b */
}

Program 2.3.8
Integrator rkc4b

double RK::ssrkc4(int neqn, double t0, double u0[],
   double h, double u[], double e[])
/*
   Function ssrkc4 computes an ODE solution by the classical
   fourth order RK method for one step along the solution
   (by calls to derv to define the ODE derivative vector).
   It also estimates the truncation error of the solution,
   and applies this estimate as a correction to the solution
   vector.
*/
Argument list

- `neqn`: number of first order ODEs
- `t0`: initial value of independent variable
- `u0`: initial condition vector of length `neqn`
- `h`: integration step
- `t`: independent variable
- `u`: ODE solution vector of length `neqn` after one rkc4 step
- `e`: estimate of truncation error of the solution vector

```c
/* Type variables */
double ut0[SIZE], ut[SIZE], u4[SIZE];
double t, k1[SIZE], k2[SIZE], k3[SIZE], k4[SIZE];
int j;

/* Derivative vector at initial (base) point */
derv(ut0,t0,u0);

/* k1; stepping for k2 */
for(j=1;j<=neqn;j++)
{
    k1[j]=h*ut0[j];
    u[j]=u0[j]+0.5*k1[j];
}
    t=t0+0.5*h;
/* Derivative vector at next RK point */
derv(ut,t,u);

/* k2; stepping for k3 */
for(j=1;j<=neqn;j++)
{
    k2[j]=h*ut[j];
    u[j]=u0[j]+0.5*k2[j];
}
    t=t0+0.5*h;
```
Program 2.3.9

Integrator ssrkc4 for a classical fourth-order RK step

double RK::rkf45a(int neqn, double t0, double tf, 
    double u[], double u0[], int nsteps)

    /*
    Function rkf45a computes an ODE solution by the fixed step RK Fehlberg 45 RK method for a series of points along the solution by repeatedly calling function ssrkf45 for a single RK Fehlberg 45 step.
Argument list

neqn    number of first order ODEs

t0      initial value of independent variable

tf      final value of independent variable

u0      initial condition vector of length neqn

nsteps  number of rkf45 steps

u       ODE solution vector of length neqn after
         nsteps steps

*/
{
    /* Type variables */
    double t, h, e[SIZE];
    int i, j;

    /* Integration step */
    h=(tf-t0)/nsteps;

    /* nsteps rkf45 steps */
    for(i=1;i<=nsteps;i++)
    {
        /* Single rkf45 step */
        t=ssrkf45(neqn,t0,u0,h,u,e);

        /* Reset base point values for next rkf45 step */
        for(j=1;j<=neqn;j++)
        {
            u0[j]=u[j];
        }
        t0=t;
    }

    /* End for */
}

return 0;

/* End of rkf45a */
}

Program 2.3.10
Integrator rkf45a
double RK::rkf45b(int neqn, double t0, double tf,
       double u[], double u0[], int nsteps,
       double abserr, double relerr)

/*
Function rkf45b computes an ODE solution by a variable step classical RK Fehlberg 45 method for a series of points along the solution by repeatedly calling function ssrkf45 for a single RK Fehlberg 45 step. The truncation error is estimated along the solution to adjust the integration step according to a specified error tolerance.

Argument list

neqn number of first order ODEs

t0 initial value of independent variable

tf final value of independent variable

u0 initial condition vector of length neqn

nsteps maximum number of rkf45 steps

abserr absolute error tolerance

relerr relative error tolerance

u ODE solution vector of length neqn after nsteps steps

*/
{
    /* Type variables */
    double h, hmin, t, e[SIZE];
    int i, nfin1;

    /* Integration step */
    h=(tf-t0)/2.0;

    /* Minimum allowable step */
    hmin=(tf-t0)/nsteps;

    /* Start integration */
    t=t0;

/* While independent variable is less than the final value, continue the integration */
while(t <= (tf*0.999))
{
    /* If the next step along the solution will go past the final value of the independent variable, set the step to the remaining distance to the final value */
    if((t+h) > tf)
    {
        h=tf-t;
    }

    /* Single rkf45 step */
    t=ssrk4(neqn,t0,u0,h,u,e);

    /* Flag for the end of the integration */
    nfin1=1;

    /* Check if any of the ODEs have violated the error criterion */
    for (i=1;i<=neqn;i++)
    {
        if(fabs(e[i]) > (fabs(u[i])*relerr+abserr))
        {
            /* Error violation, so integration is incomplete. Reduce integration step because of error violation and repeat integration from base point */
            nfin1=0;
            h=h/2.0;

            /* If the current step is less than the minimum allowable step, set the step to the minimum allowable value and continue integration from new base point */
            if(h < hmin)
            {
                h = hmin;
                nfin1=1;
            }
        
            break;
        }
    }
}
/* If there is no error violation, continue the integration from the new base point */
if(nfin1 == 1)
{
    for(i=1;i<=neqn;i++)
    {
        u0[i]=u[i];
    }
    t0=t;

    /* Test if integration step can be increased */
    for(i=1;i<=neqn;i++)
    {
        if(fabs(e[i]) > ((fabs(u[i])*relerr +abserr)/32.0) )
        {
            /* Integration step cannot be increased */
            nfin1=0;
            break;
        }
    }

    /* Increase integration step */
    if(nfin1 == 1)
    {
        h=h*2.0;
    }

    /* End if */
}

/* End while */
}

return 0;

/* End of rkf45b */
}

Program 2.3.11
Integrator rkf45b

double RK::ssrkf45(int neqn, double t0, double u0[],
                    double h, double u[], double e[])

Copyright 2004 by Chapman & Hall/CRC
Function ssrkf45 computes an ODE solution by the RK Fehlberg 45 method for one step along the solution (by calls to derv to define the ODE derivative vector). It also estimates the truncation error of the solution, and applies this estimate as a correction to the solution vector.

Argument list

neqn number of first order ODEs

t0 initial value of independent variable

u0 initial condition vector of length neqn

h integration step

t independent variable

u ODE solution vector of length neqn after one rkf45 step

e estimate of truncation error of the solution vector

*/
{
    /* Type variables */
    double t, ut0[SIZE], ut[SIZE], u5[SIZE];
    double k1[SIZE], k2[SIZE], k3[SIZE], k4[SIZE], k5[SIZE], k6[SIZE];
    int j;

    /* Derivative vector at initial (base) point */
derv(ut0,t0,u0);

    /* k1; stepping for k2 */
    for(j=1;j<=neqn;j++)
    {
        k1[j]=h*ut0[j];
        u[j]=u0[j]+0.25*k1[j];
    }
    t=t0+0.25*h;
/* Derivative vector at next RK point */
derv(ut,t,u);

/* k2; stepping for k3 */
for(j=1;j<=neqn;j++)
{
    k2[j]=h*ut[j];
    u[j]=u0[j]+(3.0/32.0)*k1[j]
       +(9.0/32.0)*k2[j];
}
t=t0+(3.0/8.0)*h;

/* Derivative vector at next RK point */
derv(ut,t,u);

/* k3; stepping for k4 */
for(j=1;j<=neqn;j++)
{
    k3[j]=h*ut[j];
    u[j]=u0[j]+(1932.0/2197.0)*k1[j]
       -(7200.0/2197.0)*k2[j]
       +(7296.0/2197.0)*k3[j];
}
t=t0+(12.00/13.0)*h;

/* Derivative vector at next RK point */
derv(ut,t,u);

/* k4; stepping for k5 */
for(j=1;j<=neqn;j++)
{
    k4[j]=h*ut[j];
    u[j]=u0[j]+( 439.0/ 216.0)*k1[j]
       -( 8.0 )*k2[j]
       +(3680.0/ 513.0)*k3[j]
       -( 845.0/4104.0)*k4[j];
}
t=t0+h;

/* Derivative vector at next RK point */
derv(ut,t,u);

/* k5; stepping for k6 */
for(j=1;j<=neqn;j++)
{


k5[j]=h*ut[j];
    u[j]=u0[j]-( 8.0/ 27.0)*k1[j]
      +( 2.0       )*k2[j]
      -(3544.0/2565.0)*k3[j]
      +(1859.0/4104.0)*k4[j]
      -( 11.0/ 40.0)*k5[j];
}
t=t0+0.5*h;

/* Derivative vector at next RK point */
derv(ut,t,u);

/* k6; fourth and fifth order step; error estimate; error correction */
for(j=1;j<=neqn;j++)
{
    k6[j]=h*ut[j];
    u[j]=u0[j]+( 25.0/ 216.0)*k1[j]
      +(1408.0/2565.0)*k3[j]
      +( 2197.0/4104.0)*k4[j]
      -( 1.0/ 5.0)*k5[j];
    u5[j]=u0[j]+( 16.0/ 135.0)*k1[j]
      +(6656.0/12825.0)*k3[j]
      +(28561.0/56430.0)*k4[j]
      -( 9.0/ 50.0)*k5[j]
      +( 2.0/ 55.0)*k6[j];
    e[j]=u5[j]-u[j];
    u[j]= u[j]+e[j];
}
t=t0+h;

return t;

/* End of ssrkf45 */
}

Program 2.3.12
Integrator ssrkf45 for an RKF45 step

/*
Routines for the RK Integration
*/

Copyright 2004 by Chapman & Hall/CRC
class RK:public DEF
{
    public:

    double rkc4a(int neqn, double t0, double tf, double u[],
                  double u0[], int nsteps);

    double rkc4b(int neqn, double t0, double tf, double u[],
                  double u0[], int nsteps, double abserr,
                  double relerr);

    double ssrkc4(int neqn, double t0, double u0[],
                  double h, double u[], double e[]);

    double rkf45a(int neqn, double t0, double tf,
                  double u[], double u0[], int nsteps);

    double rkf45b(int neqn, double t0, double tf,
                  double u[], double u0[], int nsteps,
                  double abserr, double relerr);

    double ssrkf45(int neqn, double t0, double u0[],
                   double h, double u[], double e[]);
};

Program 2.3.13
Header file for rkc4a, rkc4b, ssrkc4, rkf45a, rkf45b, ssrkf45

#include "DEF.h"
#include <iomanip.h>

/* Define file ID */
FILE *fid;

void DEF::intpar()

/* Function intpar sets the parameters to control the
integration of the 1 x 1 ODE system */
{
    /* Number of ODEs */
    neqn=1;

    /* Number of output points */
    nout=6;
/ * Maximum number of steps in the interval t0 to tf */
  nsteps=100;

/* Initial, final values of independent variable */
t0=0.0;
tf=1.0;

/* Error tolerances */
abserr=pow(10.0,-5.0);
relerr=pow(10.0,-5.0);

/* End of intpar */
}

void DEF::initial()

/* Function initial sets the initial condition vector for
the 1 x 1 ODE problem */
{
  /* Initial condition */
  u0[1]=1.0;

  /* End of initial */
}

void DEF::derv(double ut[], double t, double u[])

/* Function deriv computes the derivatives vector of the
1 x 1 ODE problem */
{
  /* Type variables */
  double alpha, lambda;

  /* Problem parameters */
  alpha=1.0;
  lambda=1.0;

  /* Derivative vector */
  ut[1]=lambda*exp(-alpha*t)*u[1];

  /* End of deriv */
}
void DEF::fprint(int ncase, int neqn, double t, double u[])

/* Function fprint displays the numerical and exact solutions to the 1 x 1 ODE problem; this routine is implemented in the traditional C style*/
{
    /* Type variables */
    double ue[2], diff[2];
    double u0, alpha, lambda;

    /* Problem parameters */
    u0=1.0;
    alpha=1.0;
    lambda=1.0;

    /* Print a heading for the solution at t = 0 */
    if(t<=0.0)
    {
        /* Label for ODE integrator */
        switch(ncase)
        {
            /*Fixed step modified Euler */
            case 1:
                fprintf(fid,"\n\n euler2a integrator\n\n");
                break;

            /* Variable step modified Euler */
            case 2:
                fprintf(fid,"\n\n euler2b integrator\n\n");
                break;

            /* Fixed step classical fourth order RK */
            case 3:
                fprintf(fid,"\n\n rkc4a integrator\n\n");
                break;

            /* Variable step classical fourth order RK */
            case 4:
                fprintf(fid,"\n\n rkc4b integrator\n\n");
                break;

            /* Fixed step RK Fehlberg 45 */
            case 5:
                fprintf(fid,"\n\n rkf45a integrator\n\n");
                break;
        }
    }
/* Variable step RK Fehlberg 45 */
case 6:
    fprintf(fid, "\n\n rkf45b integrator\n\n");
    break;

} /* Heading */
    fprintf(fid, "\n t u1(num) u1(ex) diff1\n\n");

/* Analytical solution */
ue[1]=u0*exp(lambda/alpha*(1.0-exp(-alpha*t)));

diff[1]=u[1]-ue[1];

/* Display the numerical and exact solutions, and their difference */
fprintf(fid,"%10.2f %10.5f %10.5f %13.4e\n\n",t,u[1],ue[1],diff[1]);

/* End of fprint */
}

void DEF::fprint(ofstream &fout, int ncase, int neqn, double t, double u[])

/* Function fprint displays the numerical and exact solutions to the 1 x 1 ODE problem; this function is implemented in the C++ style */
{
    /* Type variables */
    double ue[2], diff[2];
    double u0, alpha, lambda;

    /* Problem parameters */
    u0=1.0;
    alpha=1.0;
    lambda=1.0;

    /* Set printing format */
    fout<<setiosflags(ios::showpoint|ios::fixed)
        <<setprecision(7);
/ * Print a heading for the solution at t = 0 */
if(t<=0.0)
{
    /* Label for ODE integrator */
    switch(ncase)
    {
        /* Fixed step modified Euler */
        case 1:
            fout<<"\n\n euler2a integrator\n";
            break;
        /* Variable step modified Euler */
        case 2:
            fout<<"\n\n euler2b integrator\n";
            break;
        /* Fixed step classical fourth order RK */
        case 3:
            fout<<"\n\n rkc4a integrator\n";
            break;
        /* Variable step classical fourth order RK */
        case 4:
            fout<<"\n\n rkc4b integrator\n";
            break;
        /* Fixed step RK Fehlberg 45 */
        case 5:
            fout<<"\n\n rkf45a integrator\n";
            break;
        /* Variable step RK Fehlberg 45 */
        case 6:
            fout<<"\n\n rkf45b integrator\n";
            break;
    }

    /* Heading */
    fout<<endl;
    fout<<" t"<<setw(18)<<"u1(num)"<<setw(11)
    <<"u1(ex)"<<setw(10)<<"diff"<<"\n";

    /* End of t = 0 heading */
}
/* Analytical solution */
ue[1]=u0*exp(lambda/alpha*(1.0-exp(-alpha*t)));

/* Difference between exact and numerical solutions */
diff[1]=u[1]-ue[1];
fout<<endl;

/* Display the numerical and exact solutions, and their difference */
fout<<setw(10)<<t<<setw(12)<<u[1]<<setw(12)<<ue[1]
<<setw(12)<<diff[1];
/* End of fprintf */
}

Program 2.3.14
intpar, initial, deriv, and fprintf called in the solution of Equations 1.3 and 1.4

The output from the preceding routines (written to file ode1x1cpp.out in Program 2.3.1) is as follows:

euler2a integrator

<table>
<thead>
<tr>
<th>t</th>
<th>u1(num)</th>
<th>u1(ex)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.000000</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.000000</td>
</tr>
<tr>
<td>1.000000</td>
<td>1.881596</td>
<td>1.881596</td>
<td>-0.000001</td>
</tr>
<tr>
<td>2.000000</td>
<td>2.374213</td>
<td>2.374209</td>
<td>0.0000039</td>
</tr>
<tr>
<td>3.000000</td>
<td>2.586266</td>
<td>2.586260</td>
<td>0.000060</td>
</tr>
<tr>
<td>4.000000</td>
<td>2.668954</td>
<td>2.668947</td>
<td>0.000069</td>
</tr>
<tr>
<td>5.000000</td>
<td>2.700035</td>
<td>2.700027</td>
<td>0.000073</td>
</tr>
</tbody>
</table>

euler2b integrator

<table>
<thead>
<tr>
<th>t</th>
<th>u1(num)</th>
<th>u1(ex)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.000000</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.000000</td>
</tr>
<tr>
<td>1.000000</td>
<td>1.881590</td>
<td>1.881596</td>
<td>-0.0000062</td>
</tr>
<tr>
<td>2.000000</td>
<td>2.374206</td>
<td>2.374209</td>
<td>-0.0000038</td>
</tr>
<tr>
<td>3.000000</td>
<td>2.586260</td>
<td>2.586263</td>
<td>0.0000033</td>
</tr>
<tr>
<td>4.000000</td>
<td>2.668951</td>
<td>2.668947</td>
<td>0.0000037</td>
</tr>
<tr>
<td>5.000000</td>
<td>2.700034</td>
<td>2.700027</td>
<td>0.0000062</td>
</tr>
</tbody>
</table>
rkc4a integrator

<table>
<thead>
<tr>
<th>t</th>
<th>u1(num)</th>
<th>u1(ex)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0000000</td>
<td>1.0000000</td>
<td>1.0000000</td>
<td>0.0000000</td>
</tr>
<tr>
<td>1.0000000</td>
<td>1.8815964</td>
<td>1.8815964</td>
<td>0.0000000</td>
</tr>
<tr>
<td>2.0000000</td>
<td>2.3742099</td>
<td>2.3742099</td>
<td>0.0000000</td>
</tr>
<tr>
<td>3.0000000</td>
<td>2.5862603</td>
<td>2.5862603</td>
<td>0.0000000</td>
</tr>
<tr>
<td>4.0000000</td>
<td>2.6689479</td>
<td>2.6689479</td>
<td>0.0000000</td>
</tr>
<tr>
<td>5.0000000</td>
<td>2.7000278</td>
<td>2.7000278</td>
<td>0.0000000</td>
</tr>
</tbody>
</table>

rkc4b integrator

<table>
<thead>
<tr>
<th>t</th>
<th>u1(num)</th>
<th>u1(ex)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0000000</td>
<td>1.0000000</td>
<td>1.0000000</td>
<td>0.0000000</td>
</tr>
<tr>
<td>1.0000000</td>
<td>1.8815963</td>
<td>1.8815964</td>
<td>0.0000000</td>
</tr>
<tr>
<td>2.0000000</td>
<td>2.3742100</td>
<td>2.3742099</td>
<td>0.0000001</td>
</tr>
<tr>
<td>3.0000000</td>
<td>2.5862604</td>
<td>2.5862603</td>
<td>0.0000001</td>
</tr>
<tr>
<td>4.0000000</td>
<td>2.6689480</td>
<td>2.6689479</td>
<td>0.0000001</td>
</tr>
<tr>
<td>5.0000000</td>
<td>2.7000279</td>
<td>2.7000278</td>
<td>0.0000001</td>
</tr>
</tbody>
</table>

rkf45a integrator

<table>
<thead>
<tr>
<th>t</th>
<th>u1(num)</th>
<th>u1(ex)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0000000</td>
<td>1.0000000</td>
<td>1.0000000</td>
<td>0.0000000</td>
</tr>
<tr>
<td>1.0000000</td>
<td>1.8815964</td>
<td>1.8815964</td>
<td>0.0000000</td>
</tr>
<tr>
<td>2.0000000</td>
<td>2.3742099</td>
<td>2.3742099</td>
<td>0.0000000</td>
</tr>
<tr>
<td>3.0000000</td>
<td>2.5862603</td>
<td>2.5862603</td>
<td>0.0000000</td>
</tr>
<tr>
<td>4.0000000</td>
<td>2.6689479</td>
<td>2.6689479</td>
<td>0.0000000</td>
</tr>
<tr>
<td>5.0000000</td>
<td>2.7000278</td>
<td>2.7000278</td>
<td>0.0000000</td>
</tr>
</tbody>
</table>

rkf45b integrator

<table>
<thead>
<tr>
<th>t</th>
<th>u1(num)</th>
<th>u1(ex)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0000000</td>
<td>1.0000000</td>
<td>1.0000000</td>
<td>0.0000000</td>
</tr>
<tr>
<td>1.0000000</td>
<td>1.8815963</td>
<td>1.8815964</td>
<td>0.0000000</td>
</tr>
<tr>
<td>2.0000000</td>
<td>2.3742100</td>
<td>2.3742099</td>
<td>0.0000001</td>
</tr>
<tr>
<td>3.0000000</td>
<td>2.5862604</td>
<td>2.5862603</td>
<td>0.0000001</td>
</tr>
<tr>
<td>4.0000000</td>
<td>2.6689480</td>
<td>2.6689479</td>
<td>0.0000001</td>
</tr>
<tr>
<td>5.0000000</td>
<td>2.7000279</td>
<td>2.7000278</td>
<td>0.0000001</td>
</tr>
</tbody>
</table>

We note again that the output verifies the operation of all six integrators. The fixed step integrators (rkc4a, rkf45a, and ssrkc4) generally surpass the error.
tolerances set in intpar, with rkf45a and ssrkc4 producing excessive accuracy; the variable step integrators (rkc4b, rkf45b, and ssrkc4) produce numerical solutions consistent with the error tolerances.

This completes the programming of the 1x1 ODE problem in C++. We now move on to Fortran.

2.4 Programming in Fortran

Again, we start with the listing of the main program and ODE integrator routines:

```
program ode1x1
C Numerical solution to the 1 x 1 ODE system by six integrators
C Double precision coding is used
implicit double precision(a-h,o-z)
C Size arrays
parameter(neq=500)
dimension u0(neq),u(neq)
C Open a file for output
no=2
open(no,file='ode1x1for.out')
C Step through six integrators
do ncase=1,6
  C Integration parameters
call intpar(neqn,nout,nsteps,t0,tf,abserr,relerr)
  C Initial condition
call inital(neqn,t0,u0)
  C Output time
tp=tf-t0
  C Step through nout grid points
do j=1,nout
    C Print solution
call fprint(no,ncase,neqn,t0,u0)
```

Copyright © 2004 by Chapman & Hall/CRC
Select ODE integrator

Fixed step modified Euler integrator
if(ncase.eq.1)then
  call euler2a(neqn,t0,tf,u0,nsteps,u)
end if

Variable step modified Euler integrator
if(ncase.eq.2)then
  call euler2b(neqn,t0,tf,u0,nsteps,abserr,relerr,u)
end if

Fixed step classical fourth order RK integrator
if(ncase.eq.3)then
  call rkc4a(neqn,t0,tf,u0,nsteps,u)
end if

Variable step classical fourth order RK integrator
if(ncase.eq.4)then
  call rkc4b(neqn,t0,tf,u0,nsteps,abserr,relerr,u)
end if

Fixed step RK Fehlberg (RKF45) integrator
if(ncase.eq.5)then
  call rkf45a(neqn,t0,tf,u0,nsteps,u)
end if

Variable step Fehlberg (RKF45) integrator
if(ncase.eq.6)then
  call rkf45b(neqn,t0,tf,u0,nsteps,abserr,relerr,u)
end if

Advance solution
  t0=tf
  tf=tf+tp
  do i=1,neqn
    u0(i)=u(i)
  end do

Next output
  end do
Program 2.4.1
Fortran main program \textit{ode1x1} for the numerical integration of Equation 1.3 subject to initial condition Equation 1.4

Note again, as in Programs 2.2.1 and 2.3.1, that all six integrators are called within a single loop.

Also, an output file, \textit{ode1x1f.out} is written by this main program. The ODE integration routines are listed below:

\begin{verbatim}
subroutine euler2a(neqn,t0,tf,u0,nsteps,u)
   C
   C Subroutine euler2a computes an ODE solution by a fixed
   C step modified Euler method for a series of points along
   C the solution by repeatedly calling subroutine sseuler for
   C a single modified Euler step.
   C
   C Argument list
   C
   C neqn number of first order ODEs
   C
t0 initial value of independent variable
   C
tf final value of independent variable
   C
   u0 initial condition vector of length neqn
   C
   nsteps number of modified Euler steps
   C
   u ODE solution vector of length neqn after
   C
   nsteps steps
   C
   C Double precision coding is used
      implicit double precision(a-h,o-z)
   C
   C Size arrays
      dimension u0(neqn), u(neqn), e(neqn)
\end{verbatim}
Integration step
   \( h = \frac{(t_f - t_0)}{d \text{float}(n\text{steps})} \)

\( n\text{steps} \) modified Euler steps
   do \( j = 1, n\text{steps} \)
   Modified Euler step
      call \text{sseuler}(\text{neqn}, t_0, u_0, h, t, u, e)
   Reset base point values for the next modified Euler step
      do \( i = 1, \text{neqn} \)
         \( u_0(i) = u(i) \)
      end do
      \( t_0 = t \)
   Next modified Euler step
   end do
\( n\text{steps} \) modified Euler steps completed
   return
End of euler2a
end

Program 2.4.2
Integrator \textit{euler2a}

\begin{verbatim}
subroutine euler2b(neqn,t0,tf,u0,nsteps,abserr,relerr,u)
  C
  C Subroutine euler2b computes an ODE solution by a variable step modified Euler method for a series of points along the solution by repeatedly calling subroutine sseuler for a single modified Euler step. The truncation error is estimated along the solution to adjust the integration step according to a specified error tolerance.
  C
  C Argument list
  C
  C  neqn   number of first order ODEs
  C
  C  t0     initial value of independent variable
\end{verbatim}
C tf final value of independent variable
C u0 initial condition vector of length neqn
C nsteps maximum number of modified Euler steps
C abserr absolute error tolerance
C relerr relative error tolerance
C u ODE solution vector of length neqn after nsteps steps
C Double precision coding is used
   implicit double precision(a-h,o-z)
C Size the arrays
   parameter(neq=500)
   dimension e(neq)
   dimension u0(neqn), u(neqn)
C Initial integration step
   h=(tf-t0)/8.0d0
C Minimum allowable step
   hmin=(tf-t0)/dfloat(nsteps)
C Start integration
   t=t0
C While independent variable is less than the final value, continue the integration
   do while(t.le.tf*0.999d0)
C If the next step along the solution will go past the final value of the independent variable, set the step to the remaining distance to the final value
   if((t+h).gt.tf)then
      h=tf-t
   end if
C Single modified Euler step
   call sseuler(neqn,t0,u0,h,t,u,e)
C  Flag for the end of the integration
   nfin1=1
C
C  Check if any of the ODEs have violated the error
C  criteria
   do i=1,neqn
      if(dabs(e(i)).gt.dabs(u(i))*relerr+abserr)then
C       Error violation, so integration is not
C       complete. Reduce integration step because
C       of error violation and repeat integration
C       from the base point
         nfin1=0
         h=h/2.0d0
C       If the current step is less than the minimum
C       allowable step, set the step to the minimum
C       allowable value and continue integration from
C       new base point
         if(h.lt.hmin)then
            h=hmin
            nfin1=1
         end if
      go to 1
   end if
end do
C
C  If there is no error violation, continue the
C  integration from new base point
1   if(nfin1.eq.1)then
      do i=1,neqn
         u0(i)=u(i)
      end do
      t0=t
C
C  Test if integration step can be increased
   do i=1,neqn
      if(dabs(e(i)).gt.(dabs(u(i))*relerr+abserr)
         /4.0d0)then
C       Integration step cannot be increased
      go to 2
   end if
   end do
C Increase integration step
   h=h*2.0d0
C
C Continue for no error violation (nfinl=1)
2   end if
C
C Continue do while
   end do
   return
C
C End of euler2b
   end

Program 2.4.3
Integrator euler2b

subroutine sseuler(neqn,t0,u0,h,t,u,e)
C
C Subroutine sseuler computes an ODE solution by the
C modified Euler method for one step along the solution
C (by calls to derv to define the ODE derivative vector).
C It also estimates the truncation error of the solution,
C and applies this estimate as a correction to the
C solution vector.
C
C Argument list
C
C neqn  number of first order ODEs
C t0    initial value of independent variable
C u0    initial condition vector of length neqn
C h     integration step
C t     independent variable
C u     ODE solution vector of length neqn after
        one modified Euler step
C e     estimate of truncation error of the solu-
        tion vector
C
C Double precision coding is used
implicit double precision(a-h,o-z)
C
C Size the arrays
parameter(neq=500)
dimension ut(neq),ut0(neq)
dimension u0(neqn), u(neqn), e(neqn)
C
C Derivative vector at initial (base) point
 call derv(neqn,t0,u0,ut0)
C
C First order (Euler) step
 do i=1,neqn
   u(i)=u0(i)+ut0(i)*h
 end do
 t=t0+h
C
C Derivative vector at advance point
 call derv(neqn,t,u,ut)
C
C Second order step
 do i=1,neqn

C Truncation error estimate
  e(i)=(ut(i)-ut0(i))*h/2.0d0
C
C Second order (modified Euler) solution vector
  u(i)=u(i)+e(i)
 end do
 return
C
C End of sseuler
end

Program 2.4.4
Integrator sseuler for a single modified Euler step

 subroutine rkc4a(neqn,t0,tf,u0,nsteps,u)
 C
 C Subroutine rkc4a computes an ODE solution by a fixed step
 C rkc4 method for a series of points along the solution by
 C repeatedly calling subroutine ssrkc4 for a single rkc4
 C step.
 C
 C Argument list
 C
 C neqn number of first order ODEs
C t0    initial value of independent variable
C tf    final value of independent variable
C u0    initial condition vector of length neqn
C nsteps number of rkc4 steps
C u     ODE solution vector of length neqn after
        nsteps steps
C
C Double precision coding is used
   implicit double precision(a-h,o-z)
C
C Size arrays
   parameter(neq=500)
   dimension  e(neq)
   dimension  u0(neqn), u(neqn)
C
C Integration step
   h=(tf-t0)/dfloat(nsteps)
C
C nsteps rkc4 steps
   do j=1,nsteps
   C Single rkc4 step
   call ssrkc4(neqn,t0,u0,h,t,u,e)
   C Reset base point values for the next rkc4 step
   do i=1,neqn
      u0(i)=u(i)
   end do
   t0=t
   C Next rkc4 step
   end do
C
C nsteps rkc4 steps completed
   return
C
C End of rkc4a
end

Program 2.4.5
Integrator rkc4a
subroutine rkc4b(neqn,t0,tf,u0,nsteps,abserr,relerr,u)

C
C Subroutine rkc4b computes an ODE solution by a variable
C step classical fourth order RK method for a series of
C points along the solution by repeatedly calling
C subroutine ssrkc4 for a single classical fourth order RK
C step. The truncation error is estimated along the
C solution to adjust the integration step according to a
C specified error tolerance.
C
C Argument list
C
C neqn number of first order ODEs
C t0 initial value of independent variable
C tf final value of independent variable
C u0 initial condition vector of length neqn
C nsteps maximum number of rkc4 steps
C abserr absolute error tolerance
C relerr relative error tolerance
C u ODE solution vector of length neqn after
C nsteps steps
C
C Double precision coding is used
   implicit double precision(a-h,o-z)
C
C Size the arrays
   parameter(neq=500)
   dimension e(neq)
   dimension u0(neqn), u(neqn)
C
C Integration step
   h=(tf-t0)/2.0d0
C
C Minimum allowable step
   hmin=(tf-t0)/dfloat(nsteps)
C
C Start integration
   t=t0
While independent variable is less than the final value, continue the integration

\[
do\;while(t.\leq tf*0.999d0)
\]

If the next step along the solution will go past the final value of the independent variable, set the step to the remaining distance to the final value

\[
if((t+h).gt.tf)then
    h=tf-t
end\;if
\]

Single rkc4 step

\[
call\;ssrkcr4(neqn,t0,u0,h,t,u,e)
\]

Flag for the end of the integration

\[
nfin1=1
\]

Check if any of the ODEs have violated the error criteria

\[
do\;i=1,neqn
    if(dabs(e(i)).gt.dabs(u(i))*relerr+abserr)then
    \]

Error violation, so integration is not complete. Reduce integration step because of error violation and repeat integration from the base point

\[
nfin1=0
    h=h/2.0d0
end\;if
\]

If the current step is less than the minimum allowable step, set the step to the minimum allowable value and continue integration from new base point

\[
if(h.lt.hmin)then
    h=hmin
    nfin1=1
end\;if
\]

If there is no error violation, continue the integration from new base point
if(nfin1.eq.1)then
    do i=1,neqn
        u0(i)=u(i)
    end do
    t0=t

  C  Test if integration step can be increased
  do i=1,neqn
      if(dabs(e(i)).gt.(dabs(u(i))*relerr+abserr)
         /16.0d0)then
          C  Integration step cannot be increased
          go to 2
          end if
  end do

  C  Increase integration step
  h=h*2.0d0

  C  Continue for no error violation (nfin1=1)
2  end if

C  Continue do while
  end do
  return

C  End of rkc4b
  end

Program 2.4.6
Integrator rkc4b

subroutine ssrkc4(neqn,t0,u0,h,t,u,e)

C  Subroutine ssrkc4 computes an ODE solution by the
C  classical fourth order RK method for one step along the
C  solution (by calls to derv to define the ODE derivative
C  vector). It also estimates the truncation error of the
C  solution, and applies this estimate as a correction
C  to the solution vector.
C
C  Argument list
C
C  neqn      number of first order ODEs

C t0 initial value of independent variable
C u0 initial condition vector of length neqn
C h integration step
C t independent variable
C u ODE solution vector of length neqn after one rkc4 step
C e estimate of truncation error of the solution vector
C
C Double precision coding is used
   implicit double precision(a-h,k,o-z)
C
C Size the arrays
parameter(neq=500)
dimension ut0(neq), ut(neq), u4(neq),
   k1(neq), k2(neq), k3(neq), k4(neq)
dimension u0(neqn), u(neqn), e(neqn)
C
C Derivative vector at initial (base) point
   call derv(neqn,t0,u0,ut0)
C
C k1, advance of dependent variable vector and independent variable for calculation of k2
   do i=1,neqn
      k1(i)=h*ut0(i)
      u(i)=u0(i)+0.5d0*k1(i)
   end do
   t=t0+0.5d0*h
C
C Derivative vector at new u, t
   call derv(neqn,t,u,ut)
C
C k2, advance of dependent variable vector and independent variable for calculation of k3
   do i=1,neqn
      k2(i)=h*ut(i)
      u(i)=u0(i)+0.5d0*k2(i)
   end do
   t=t0+0.5d0*h
C
C Derivative vector at new u, t
    call derv(neqn,t,u,ut)

C k3, advance of dependent variable vector and
C independent variable for calculation of k4
    do i=1,neqn
        k3(i)=h*ut(i)
        u(i)=u0(i)+k3(i)
    end do
    t=t0+h

C Derivative vector at new u, t
    call derv(neqn,t,u,ut)

C k4, stepping
    do i=1,neqn
        k4(i)=h*ut(i)
    end do

C Second order step
    u(i)=u0(i)+k2(i)

C Fourth order step
    u4(i)=u0(i)+(1.0d0/6.0d0)*(k1(i)+2.0d0*k2(i)
        +2.0d0*k3(i)+k4(i))
    end do
    do i=1,neqn

C Truncation error estimate
    e(i)=u4(i)-u(i)

C Fourth order solution vector (from (2,4) RK pair)
    u(i)=u(i)+e(i)
    end do
    return

C End of ssrkc4
end

Program 2.4.7
Integrator ssrkc4 for a classical fourth-order RK step

    subroutine rkf45a(neqn,t0,tf,u0,nsteps,u)

    C Subroutine rkf45a computes an ODE solution by a fixed
    C step rkf45 method for a series of points along the
C solution by repeatedly calling subroutine ssrkf45 for
C a single rkf45 step.
C
C Argument list
C
C neqn number of first order ODEs
C
t0 initial value of independent variable
C
tf final value of independent variable
C
C u0 initial condition vector of length neqn
C
C nsteps number of rkf45 steps
C
C u ODE solution vector of length neqn after
C nsteps steps
C
C Double precision coding is used
   implicit double precision(a-h,o-z)
C
C Size arrays
   parameter(neq=500)
   dimension  e(neq)
   dimension  u0(neqn), u(neqn)
C
C Integration step
   h=(tf-t0)/dfloat(nsteps)
C
C nsteps rkf45 steps
   do j=1,nsteps
C
C Single rkf45 step
   call ssrkf45(neqn,t0,u0,h,t,u,e)
C
C Reset base point values for the next rkf45 step
   do i=1,neqn
      u0(i)=u(i)
   end do
   t0=t
C
C Next rkf45 step
   end do
C nsteps rkf45 steps completed
   return
C
C End of rkf45a
end

Program 2.4.8
Integrator rkf45a

subroutine rkf45b(neqn,t0,tf,u0,nsteps,abserr,relerr,u)
C
C Subroutine rkf45b computes an ODE solution by a variable
C step RK Fehlberg method for a series of points along the
C solution by repeatedly calling subroutine ssrkf45 for a
C single RK Fehlberg step. The truncation error is
C estimated along the solution to adjust the integration
C step according to a specified error tolerance.
C
C Argument list
C
C neqn number of first order ODEs
C t0 initial value of independent variable
C tf final value of independent variable
C u0 initial condition vector of length neqn
C nsteps maximum number of rkf45 steps
C abserr absolute error tolerance
C relerr relative error tolerance
C u ODE solution vector of length neqn after
C nsteps steps
C
C Double precision coding is used
   implicit double precision(a-h.o-z)
C
C Size the arrays
   parameter(neq=500)
   dimension   e(neq)
   dimension u0(neqn), u(neqn)
C Initial integration step
   h=(tf-t0)/2.0d0
C
C Minimum allowable step
   hmin=(tf-t0)/dfloat(nsteps)
C
C Start integration
   t=t0
C
C While independent variable is less than the final
C value, continue the integration
   do while(t.le.tf*0.999d0)
C
C If the next step along the solution will go past
C the final value of the independent variable, set
C the step to the remaining distance to the final
C value
   if((t+h).gt.tf)then
      h=tf-t
   end if
C
C Single rkf45 step
   call ssrkf45(neqn,t0,u0,h,t,u,e)
C
C Flag for the end of the integration
   nfin1=1
C
C Check if any of the ODEs have violated the error
C criteria
   do i=1,neqn
      if(dabs(e(i)).gt.dabs(u(i))*relerr+abserr)then
         Error violation, so integration is not
         complete. Reduce integration step because
         of error violation and repeat integration
         from the base point
         nfin1=0
         h=h/2.0d0
      end if
C
C If the current step is less than the minimum
C allowable step, set the step to the minimum
C allowable value and continue integration from
C new base point
   if(h.lt.hmin)then
      h=hmin
   end if
nfin1=1
end if
go to 1
end if
end do

C If there is no error violation, continue the
C integration from new base point
1 if(nfin1.eq.1)then
do i=1,neqn
   u0(i)=u(i)
end do
t0=t

C Test if integration step can be increased
do i=1,neqn
   if(dabs(e(i)).gt.(dabs(u(i))*relerr+abserr)/32.0d0)then
      C Integration step cannot be increased
      go to 2
   end if
end do

C Increase integration step
h=h*2.0d0

C Continue for no error violation (nfin1=1)
2 end if

C Continue do while
   end do
   return

C End of rkf45b
end

Program 2.4.9
Integrator rkf45b

subroutine ssrkf45(neqn,t0,u0,h,t,u,e)
C Subroutine ssrkf45 computes an ODE solution by the RK
C Fehlberg 45 method for one step along the solution (by
C calls to deriv to define the ODE derivative vector).
It also estimates the truncation error of the solution, and applies this estimate as a correction to the solution vector.

Argument list

- `neqn`: number of first order ODEs
- `t0`: initial value of independent variable
- `u0`: initial condition vector of length `neqn`
- `h`: integration step
- `t`: independent variable
- `u`: ODE solution vector of length `neqn` after one rkf45 step
- `e`: estimate of truncation error of the solution vector

Double precision coding is used

```
implicit double precision(a-h,k,o-z)
```

Size the arrays

```
parameter(neq=500)
dimension ut0(neq), ut(neq), u5(neq),
+ k1(neq), k2(neq), k3(neq),
+ k4(neq), k5(neq), k6(neq)
dimension u0(neqn), u(neqn), e(neqn)
```

Derivative vector at initial (base) point

```
call deriv(neqn,t0,u0,ut0)
```

`k1`, advance of dependent variable vector and independent variable for calculation of `k2`

```
do i=1,neqn
   k1(i)=h*ut0(i)
   u(i)=u0(i)+0.25d0*k1(i)
end do
```

```
t=t0+0.25d0*h
```

Derivative vector at new `u`, `t`

```
call deriv(neqn,t,u,ut)
```
C k2, advance of dependent variable vector and
C independent variable for calculation of k3
  do i=1,neqn
    k2(i)=h*ut(i)
    u(i)=u0(i)+(3.0d0/32.0d0)*k1(i)
    +      +(9.0d0/32.0d0)*k2(i)
  end do
  t=t0+(3.0d0/8.0d0)*h
C
C Derivative vector at new u, t
  call derv(neqn,t,u,ut)
C
C k3, advance of dependent variable vector and
C independent variable for calculation of k4
  do i=1,neqn
    k3(i)=h*ut(i)
    u(i)=u0(i)+(1932.0d0/2197.0d0)*k1(i)
    +      - (7200.0d0/2197.0d0)*k2(i)
    +      + (7296.0d0/2197.0d0)*k3(i)
  end do
  t=t0+(12.0d0/13.0d0)*h
C
C Derivative vector at new u, t
  call derv(neqn,t,u,ut)
C
C k4, advance of dependent variable vector and
C independent variable for calculation of k5
  do i=1,neqn
    k4(i)=h*ut(i)
    u(i)=u0(i)+ ( 439.0d0 / 216.0d0)*k1(i)
    +      - ( 8.0d0 )*k2(i)
    +      + (3680.0d0 / 513.0d0)*k3(i)
    +      - ( 845.0d0 / 4104.0d0)*k4(i)
  end do
  t=t0+h
C
C Derivative vector at new u, t
  call derv(neqn,t,u,ut)
C
C k5, advance of dependent variable vector and
C independent variable for calculation of k6
  do i=1,neqn
    k5(i)=h*ut(i)


```fortran
  u(i)=u0(i)-( 8.0d0/ 27.0d0)*k1(i)
  + +( 2.0d0       )*k2(i)
  + -(3544.0d0/2565.0d0)*k3(i)
  + +(1859.0d0/4104.0d0)*k4(i)
  + -( 11.0d0/ 40.0d0)*k5(i)
end do

  t=t0+0.5d0*h

C Derivative vector at new u, t
  call derv(neqn,t,u,ut)
C
C k6, stepping
  do i=1,neqn
    k6(i)=h*ut(i)
  end do
C
C Fourth order step
  u(i)=u0(i)+( 25.0d0 / 216.0d0)*k1(i)
  + +(1408.0d0 / 2565.0d0)*k3(i)
  + +(2197.0d0 / 4104.0d0)*k4(i)
  + -( 1.0d0 /  5.0d0)*k5(i)
C
C Fifth order step
  u5(i)=u0(i)+( 16.0d0/ 135.0d0)*k1(i)
  + +( 6656.0d0/12825.0d0)*k3(i)
  + +(28561.0d0/56430.0d0)*k4(i)
  + -( 9.0d0/  50.0d0)*k5(i)
  + +( 2.0d0/  55.0d0)*k6(i)
end do
  do i=1,neqn
C
C Truncation error estimate
  e(i)=u5(i)-u(i)
C
C Fifth order solution vector (from (4,5) RK pair)
  u(i)=u(i)+e(i)
end do

  t=t0+h
  return

C End of ssrkf45
end

Program 2.4.10
Integrator ssrkf45 for an RKF45 step
```

Copyright 2004 by Chapman & Hall/CRC
subroutine intpar(neqn,nout,nsteps,t0,tf,abserr,relerr)
C
C Subroutine intpar sets the parameters to control the
C integration of the 1 x 1 ODE system
C
C Double precision coding is used
   implicit double precision(a-h,o-z)
C
C Number of ODEs
   neqn=1
C
C Number of output points
   nout=6
C
C Maximum number of steps in the interval t0 to tf
   nsteps=100
C
C Initial, final values of the independent variable
   t0=0.0d0
   tf=1.0d0
C
C Error tolerances
   abserr=1.0d-05
   relerr=1.0d-05
   return
C
C End of intpar
   end

subroutine inital(neqn,t,u0)
C
C Subroutine inital sets the initial condition vector
C for the 1 x 1 ODE problem
C
C Double precision coding is used
   implicit double precision(a-h,o-z)
C
C Size the arrays
   dimension u0(neqn)
C
C Initial condition
   u0(1)=1.0d0
   return
C
C End of initial
end

subroutine derv(neqn,t,u,ut)
C
C Subroutine derv computes the derivative vector
C of the 1 x 1 ODE problem
C
C Double precision coding is used
    implicit double precision(a-h,l,o-z)
C
C Size the arrays
    dimension u(neqn), ut(neqn)
C
C Problem parameters
    alpha=1.0d0
    lambda=1.0d0
C
C Derivative vector
    ut(1)=lambda*dexp(-alpha*t)*u(1)
    return
C
C End of derv
end

subroutine fprint(no,ncase,neqn,t,u)
C
C Subroutine fprint displays the numerical and
C analytical solutions to the 1 x 1 ODE problem
C
C Double precision coding is used
    implicit double precision(a-h,l,o-z)
C
C Size the arrays
    dimension u(neqn)
C
C Problem parameters
    u0=1.0d0
    alpha=1.0d0
    lambda=1.0d0
C
C Print a heading for the solution at t = 0
    if(t.le.0.0d0)then
C
C Label for ODE integrator
C
C Fixed step modified Euler
if(ncase.eq.1)then
  write(no,11)
  11 format(/,6x,'euler2a integrator')
C
C Variable step modified Euler
else if(ncase.eq.2)then
  write(no,12)
  12 format(/,6x,'euler2b integrator')
C
C Fixed step classical fourth order RK
else if(ncase.eq.3)then
  write(no,13)
  13 format(/,6x,'rkc4a integrator')
C
C Variable step classical fourth order RK
else if(ncase.eq.4)then
  write(no,14)
  14 format(/,6x,'rkc4b integrator')
C
C Fixed step RK Fehlberg 45
else if(ncase.eq.5)then
  write(no,15)
  15 format(/,6x,'rkf45a integrator')
C
C Variable step RK Fehlberg 45
else if(ncase.eq.6)then
  write(no,16)
  16 format(/,6x,'rkf45b integrator')
end if
C
C Heading
  write(no,2)
  2 format(/,9x,'t',3x,'u1(num)',4x,'u1(ex)',8x,'diff1',/)
C
C End of t = 0 heading
  end if
C
C Analytical solution
  uexact=u0*dexp(lambda/alpha*(1.0d0-dexp(-alpha*t)))
C
C Difference between exact and numerical solution vectors
  diff1=u(1)-uexact
C
C Display the numerical and exact solutions, and their
difference
write(no,3)t,u(1),uexact,diff1
3 format(f10.2,2f10.5,e13.4)
return
C
C End of fprint
end

Program 2.4.11

The output from the preceding routines (written to file ode1x1for.out in Program 2.4.1) is as follows:

euler2a integrator

<table>
<thead>
<tr>
<th>t</th>
<th>u1(num)</th>
<th>u1(ex)</th>
<th>diff1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.00000</td>
<td>1.00000</td>
<td>0.0000E+00</td>
</tr>
<tr>
<td>1.00</td>
<td>1.88160</td>
<td>1.88160</td>
<td>-0.6467E-07</td>
</tr>
<tr>
<td>2.00</td>
<td>2.37421</td>
<td>2.37421</td>
<td>0.3879E-05</td>
</tr>
<tr>
<td>3.00</td>
<td>2.58627</td>
<td>2.58626</td>
<td>0.6033E-05</td>
</tr>
<tr>
<td>4.00</td>
<td>2.66895</td>
<td>2.66895</td>
<td>0.6924E-05</td>
</tr>
<tr>
<td>5.00</td>
<td>2.70004</td>
<td>2.70003</td>
<td>0.7265E-05</td>
</tr>
</tbody>
</table>

euler2b integrator

<table>
<thead>
<tr>
<th>t</th>
<th>u1(num)</th>
<th>u1(ex)</th>
<th>diff1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.00000</td>
<td>1.00000</td>
<td>0.0000E+00</td>
</tr>
<tr>
<td>1.00</td>
<td>1.88159</td>
<td>1.88160</td>
<td>-0.6220E-05</td>
</tr>
<tr>
<td>2.00</td>
<td>2.37421</td>
<td>2.37421</td>
<td>-0.3778E-05</td>
</tr>
<tr>
<td>3.00</td>
<td>2.58626</td>
<td>2.58626</td>
<td>0.2933E-06</td>
</tr>
<tr>
<td>4.00</td>
<td>2.66895</td>
<td>2.66895</td>
<td>0.3665E-05</td>
</tr>
<tr>
<td>5.00</td>
<td>2.70003</td>
<td>2.70003</td>
<td>0.6249E-05</td>
</tr>
</tbody>
</table>

rkc4a integrator

<table>
<thead>
<tr>
<th>t</th>
<th>u1(num)</th>
<th>u1(ex)</th>
<th>diff1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.00000</td>
<td>1.00000</td>
<td>0.0000E+00</td>
</tr>
<tr>
<td>1.00</td>
<td>1.88160</td>
<td>1.88160</td>
<td>-0.2719E-10</td>
</tr>
</tbody>
</table>
We note again that the output verifies the operation of all six integrators. The fixed step integrators (*rkc4a*, *rkf45a*, and *ssrkc4*) generally surpass the error tolerances set in *intpar*, with *rkf45a* and *ssrkc4* producing excessive accuracy;
the variable step integrators (rkc4b, rkf45b, and ssrkc4) produce numerical solutions consistent with the error tolerances.

This completes the programming of the 1x1 ODE problem in Fortran. We now move on to Java.

2.5 Programming in Java

As before, we start with the listing of the main program and ODE integrator routines (intpar, initial, derv, and fprint are copied to DEF.java before execution):

```java
/*
Numerical solution to the 1 x 1 ODE system by six integrators
*/

/* Include libraries */
import java.io.*;
import java.text.*;
import java.math.*;
import mol.*;

/* Main program */
public class ode1x1m{
    public static void main(String[] args){
        /* Type variables */
        double tp;
        int i, j, ncase;

        /* Declare an object from DEF */
        DEF o1;

        /* Instantiate a class from Euler */
        Euler el=new Euler();

        /* Instantiate a class from RK */
        RK r1=new RK();
    }
}
```
/* Use a buffer stream for the solution */ PrintWriter f=null;

/* Open a file for output */
try {
    f = new PrintWriter(new FileOutputStream
       ("ode1x1jv.out"), true);
} catch(IOException ioe) {
    System.out.println("Attempt to open file for
        writing failed");
    System.exit(-1);
}

/* Step through six integrators */
for(ncase=1;ncase<=6;ncase++) {
    /* Integration parameters, initial condition */
    o1=new DEF();
    /* Output interval */
    tp=o1.tf-o1.t0;

    /* Complete solution at nout output points */
    for(j=1;j<=o1.nout;j++) {
        /* Print current solution */
        o1.fprint(f,ncase,o1.neqn,o1.t0,o1.u0);

        /* Select ODE integrator */
        switch(ncase) {
        case 1:
            e1.euler2a(o1.neqn,o1.t0,o1.tf,o1.u,o1.u0,
                o1.nsteps);
            break;
        case 2:
            e1.euler2b(o1.neqn,o1.t0,o1.tf,o1.u,o1.u0,
                o1.nsteps, o1.abserr, o1.relerr);
            break;
        }
/* Fixed step classical fourth order RK integrator */
case 3:
r1.rkc4a(o1.neqn,o1.t0,o1.tf,o1.u,o1.u0,
o1.nsteps);
break;

/* Variable step classical fourth order RK integrator */
case 4:
r1.rkc4b(o1.neqn,o1.t0,o1.tf,o1.u,o1.u0,
o1.nsteps, o1.abserr, o1.relerr);
break;

/* Fixed step RK Fehlberg (RKF45) integrator */
case 5:
r1.rkf45a(o1.neqn,o1.t0,o1.tf,o1.u,o1.u0,
o1.nsteps);
break;

/* Variable step RK Fehlberg (RKF45) integrator */
case 6:
r1.rkf45b(o1.neqn,o1.t0,o1.tf,o1.u,o1.u0,
o1.nsteps, o1.abserr, o1.relerr);
break;

} /* Next integrator */

/* Advance solution */
o1.t0=o1.tf;
o1.tf+=tp;
for(i=1;i<=o1.neqn;i++)
{
o1.u0[i]=o1.u[i];
}

/* Next output */

/* Next integrator */

/* Complete solution computed. Close the file stream */
f.close();
}
/* End of ode1x1m */
}

Program 2.5.1
Java main program ode1x1m for the numerical integration of Equation 1.3 subject to initial condition Equation 1.4

Note again, as in Program 2.4.1, that all six integrators are called within a single loop.

/* Step through six integrators */
for(ncase=1;ncase<=6;ncase++)

Also, an output file, ode1x1jv.out is written by this main program.
The associated interface routines are as follows:

/*
Definition of functions intpar, initial, derv, fprint
for the 1 x 1 ODE system
*/

package mol;
import java.io. *
;

public interface ode1x1interface
{

/* Integration parameters */
public void intpar();

/* Initial condition */
public void initial();

/* Derivative vector */
public void derv(double ut[], double t, double u[]);
}
/* Output */
public void fprint(PrintWriter f, int ncase, int neqn,
                      double t, double u[]);
}
/*
   Define the common (global) variables for the
   1 x 1 ODE problem
*/
package mol;

public class MOL
{
    /* Maximum (default) number of ODEs */
    public int SIZE=500;
    /* Variables for ODE integration */
    public int neqn, nout, nsteps;
    public double t0, tf, abserr, relerr;
    public double u[], u0[], e[];
    public MOL()
    {
    }
}

Program 2.5.2
Interface routines used by Program 2.5.1

The ODE integration routines are listed below:

package mol;

import java.math.*;
import java.io.*;
public class Euler extends DEF implements EulerInterface
{

    public Euler()
    {
    }

    public void euler2a(int neqn, double t0, double tf,
                        double u[], double u0[], int nsteps)
    {
        /* Type variables */
        double ut0[], ut[];
        double h, t;
        int i, j;

        /* Integration step */
        h = (tf - t0) / nsteps;

        /* nsteps modified Euler steps */
        for (i = 1; i <= nsteps; i++)
        {
            /* Single modified Euler step */
            t = sseuler(neqn, t0, u0, h, u, e);
        }
    }

    /* Function euler2a computes an ODE solution by the fixed
    step modified Euler method for a series of points along
    the solution by repeatedly calling function sseuler for
    a single modified Euler step.

    Argument list
    neqn    number of first order ODEs
    t0      initial value of independent variable
    tf      final value of independent variable
    u0      initial condition vector of length neqn
    nsteps  number of modified Euler steps
    u       ODE solution vector of length neqn after
            nsteps steps
    */

    Copyright © 2004 by Chapman & Hall/CRC
/ * Reset base point values for next modified Euler step */
   for(j=1;j<=neqn;j++)
     {
       u0[j]=u[j];
     }
   t0=t;
   /* End for */

   /* End of euler2a */
}

Program 2.5.3
Integrator euler2a

  public void euler2b(int neqn, double t0, double tf,
                     double u[], double u0[], int nsteps,
                     double abserr, double relerr)
  {
    /*
      Function euler2b computes an ODE solution by the variable
      step modified Euler method for a series of points along the
      solution by repeatedly calling function sseuler for a single
      modified Euler step. The truncation error is estimated along
      the solution to adjust the integration step according to a
      specified error tolerance.
    */

    Argument list

      neqn    number of first order ODEs
      t0      initial value of independent variable
      tf      final value of independent variable
      u0      initial condition vector of length neqn
      nsteps  number of modified Euler steps
      u       ODE solution vector of length neqn after nsteps steps
    */
double e[] = new double[SIZE];
double hmin, h, t;
int i, nfin1;

/* Integration step */
h = (tf - t0) / 8.0;

/* Minimum allowable step */
hmin = (tf - t0) / (float)(nsteps);

/* Start integration */
t = t0;

/* While independent variable is less than the final value, continue the integration */
while (t <= tf * 0.999)
{
    /* If the next step along the solution will go past the final value of the independent variable, set the step to the remaining distance to the final value */
    if ((t + h) > tf)
    {
        h = tf - t;
    }

    /* Single modified Euler step */
t = sseuler(neqn, t0, u0, h, u, e);

    /* Flag for the end of the integration */
nfin1 = 1;

    /* Check if any of the ODEs have violated the error criterion */
    for (i = 1; i <= neqn; i++)
    {
        if (Math.abs(e[i]) > (Math.abs(u[i]) * relerr + abserr))
        {
            /* Error violation, so integration is incomplete. Reduce integration step because of error violation and repeat integration from base point */
            
            nfin1 = 0;
            h = h / 2.0;
        }
/* If the current step is less than the minimum allowable step, set the step to the minimum allowable value and continue integration from new base point */
if( h < hmin )
{
    h=hmin;
    nfin1=1;
}
break;
}

/* If there is no error violation, continue the integration from the new base point */
if(nfin1 == 1 )
{
    for(i=1;i<=neqn;i++)
    {
        u0[i]=u[i];
    }
t0=t;

    /* Test if integration step can be increased */
    for(i=1;i<=neqn;i++)
    {
        if( Math.abs(e[i]) > ((Math.abs(u[i]) *relerr+abserr)/4.0) )
        {
            /* Integration step cannot be increased */
            nfin1=0;
            break;
        }
    }

    /* Increase integration step */
    if(nfin1 == 1 )
    {
        h=h*2.0;
    }

    /* End if */
}
*/ End while */

} /* End of euler2b */

Program 2.5.4
Integrator euler2b

public double sseuler(int neqn, double t0, double u0[],
                        double h, double u[], double e[])
{
    /* Function sseuler computes an ODE solution by the modified Euler method for one step along the solution (by calls to derv to define the ODE derivative vector). It also estimates the truncation error of the solution, and applies this estimate as a correction to the solution vector. */

    Argument list
    neqn number of first order ODEs
    t0 initial value of independent variable
    u0 initial condition vector of length neqn
    h integration step
    t independent variable
    u ODE solution vector of length neqn after one modified Euler step
    e estimate of truncation error of the solution vector

    */

    /* Type variables */
    double ut0[], ut[];
    double t;
    int j;
/ * Declare arrays */
  ut0=new double[SIZE];
  ut=new double[SIZE];

  /* Derivative vector at initial (base) point */
  derv(ut0,t0,u0);

  /* First order (Euler) step */
  for(j=1;j<=neqn;j++)
  {
    u[j]=u0[j]+ut0[j]*h;
  }
  t=t0+h;

  /* Derivative at advance point */
  derv(ut,t,u);

  /* Second order (modified Euler) step */
  for(j=1;j<=neqn;j++)
  {
    /* Truncation error estimate */
    e[j]=(ut[j]-ut0[j])*h/2.0;
    /* Second order (modified Euler) solution vector */
    u[j]=u[j]+e[j];
  }

  return t;

  /* End of sseuler */
  }

  /* End of public class Euler */
  }

Program 2.5.5
Integrator sseuler for a single modified Euler step

/*

Routines for the Euler ODE Integration
*/
package mol;
import java.math.*;
import java.io.*;

public interface EulerInterface
{
    /* Fixed step modified Euler method */
    public void euler2a(int neqn, double t0, double tf,
                        double u[], double u0[], int nsteps);

    /* Variable step modified Euler method */
    public void euler2b(int neqn, double t0, double tf,
                        double u[], double u0[], int nsteps,
                        double abserr, double relerr);

    /* Single step modified Euler method */
    public double sseuler(int neqn, double t0, double u0[],
                           double h, double u[], double e[]);
}

Program 2.5.6
Interface routine for euler2a, euler2b, and sseuler

package mol;
import java.io.*;
import java.math.*;

public class RK extends DEF implements RKInterface
{
    public RK()
    {
    }

    public void rkc4a(int neqn, double t0, double tf,
                       double u[], double u0[], int nsteps)
    {
        /* Function rkc4a computes an ODE solution by the fixed step
           classical fourth order RK method for a series of points
           along the solution by repeatedly calling function ssrk4c
           for a single classical fourth order RK step. */
    }
}
Argument list

neqn  number of first order ODEs

t0    initial value of independent variable

tf    final value of independent variable

u0    initial condition vector of length neqn

nsteps  number of RKC4 steps

u    ODE solution vector of length neqn after
     nsteps steps

*/
{
  /* Type variables */
  double h, t;
  double e[] = new double[SIZE];
  int i, j;

  /* Integration step */
  h=(tf-t0)/nsteps;

  /* nsteps rkc4 steps */
  for(i=1;i<=nsteps;i++)
  {
    /* Single rkc4 step */
    t=ssrkc4(neqn,t0,u0,h,u,e);

    /* Reset base point values for next rk4c step */
    for(j=1;j<=neqn;j++)
    {
      u0[j]=u[j];
    }
    t0=t;
  }

  /* End of rkc4a */
}

Program 2.5.7
Integrator rkc4a
public void rkc4b(int neqn, double t0, double tf,
        double u[], double u0[], int nsteps,
        double abserr, double relerr)

/*
 * Function rkc4b computes an ODE solution by a variable
 * step classical fourth order RK method for a series of
 * points along the solution by repeatedly calling function
 * ssrkc4 for a single classical fourth order RK step. The
 * truncation error is estimated along the solution to
 * adjust the integration step according to a specified
 * error tolerance.
 *
 * Argument list
 *
 * neqn number of first order ODEs
 * t0 initial value of independent variable
 * tf final value of independent variable
 * u0 initial condition vector of length neqn
 * nsteps maximum number of rkc4 steps
 * abserr absolute error tolerance
 * relerr relative error tolerance
 * u ODE solution vector of length neqn after
 *     nsteps steps
 */
{
/* Type variables */
    double hmin, h, t;
    double e[] = new double[SIZE];
    int nfin1, i;

/* Integration step */
    h=(tf-t0)/2.0;

/* Minimum allowable step */
    hmin=(tf-t0)/nsteps;

/* Start integration */
    t=t0;
/* While independent variable is less than the final value, continue the integration */
while(t <= (tf*0.999))
{
    /* If the next step along the solution will go past the final value of the independent variable, set the step to the remaining distance to the final value */
    if((t+h) > tf )
    {
        h=tf-t;
    }

    /* Single rkc4 step */
    t=ssrkc4(neqn,t0,u0,h,u,e);

    /* Flag for the end of the integration */
    nfin1=1;

    /* Check if any of the ODEs have violated the error criterion */
    for(i=1;i<=neqn;i++)
    {
        if(Math.abs(e[i]) > (Math.abs(u[i])*relerr+abserr))
        {
            /* Error violation, so integration is incomplete. Reduce integration step because of error violation and repeat integration from base point */
            nfin1=0;
            h=h/2.0;

            /* If the current step is less than the minimum allowable step, set the step to the minimum allowable value and continue integration from new base point */
            if(h < hmin)
            {
                h=hmin;
                nfin1=1;
            }
            break;
        }
    }

    /* If there is no error violation, continue the integration from the new base point */
if(nfin1 == 1)
{
    for(i=1;i<=neqn;i++)
    {
        u0[i]=u[i];
    }
    t0=t;

    /* Test if integration step can be increased */
    for(i=1;i<=neqn;i++)
    {
        if(Math.abs(e[i]) > ((Math.abs(u[i])
            *relerr+abserr)/16.0))
        {
            /* Integration step cannot be increased */
            nfin1=0;
            break;
        }
    }

    /* Increase integration step */
    if(nfin1 == 1)
    {
        h=h*2.0;
    }

    /* End if */
}

/* End while */

/* End of rkc4b */
}

Program 2.5.8
Integrator rkc4b

public double ssrkc4(int neqn, double t0, double u0[],
        double h, double u[], double e[])
/*
    Function ssrkc4 computes an ODE solution by the classical
    fourth order RK method for one step along the solution
    (by calls to derv to define the ODE derivative vector).
    It also estimates the truncation error of the solution,
and applies this estimate as a correction to the solution vector.

Argument list

neqn number of first order ODEs

t0 initial value of independent variable

u0 initial condition vector of length neqn

h integration step

t independent variable

u ODE solution vector of length neqn after one rkc4 step

e estimate of truncation error of the solution vector

/*
{ */
   /* Type variables */
   double ut0[], ut[], u4[];
   double t, k1[], k2[], k3[], k4[];
   int j;

   /* Declare arrays */
   ut0=new double[SIZE];
   ut=new double[SIZE];
   u4=new double[SIZE];
   k1=new double[SIZE];
   k2=new double[SIZE];
   k3=new double[SIZE];
   k4=new double[SIZE];

   /* Derivative vector at initial (base) point */
   derv(ut0,t0,u0);

   /* k1; stepping for k2 */
   for(j=1; j<=neqn; j++)
   {
      k1[j]=h*ut0[j];
      u[j]=u0[j]+0.5*k1[j];
   }
   t=t0+0.5*h;
/* Derivative vector at next RK point */
derv(ut,t,u);

/* k2; stepping for k3 */
for(j=1;j<=neqn;j++)
{
    k2[j]=h*ut[j];
    u[j]=u0[j]+0.5*k2[j];
}
t=t0+0.5*h;

/* Derivative vector at next RK point */
derv(ut,t,u);

/* k3; stepping for k4 */
for(j=1;j<=neqn;j++)
{
    k3[j]=h*ut[j];
    u[j]=u0[j]+k3[j];
}
t=t0+h;

/* Derivative vector at next RK point */
derv(ut,t,u);

/* k4; second and fourth order step; error estimate; error correction */
for(j=1;j<=neqn;j++)
{
    k4[j]=h*ut[j];
    u[j]=u0[j]+k2[j];
    u4[j]=u0[j]+(1.0/6.0)*(k1[j]+2.0*k2[j]+2.0*k3[j]
                        +k4[j]);
    e[j]=u4[j]-u[j];
    u[j]=u[j]+e[j];
}
t=t0+h;

return t;

/* End of ssrkc4 */

Program 2.5.9
Integrator ssrkc4 for a classical fourth-order RK step
public void rkf45a(int neqn, double t0, double tf, 
    double u[], double u0[], int nsteps)

/*
 Function rkf45a computes an ODE solution by the fixed
 step RK Fehlberg 45 RK method for a series of points
 along the solution by repeatedly calling function ssrkf45
 for a single RK Fehlberg 45 step.

 Argument list
 neqn number of first order ODEs
 t0 initial value of independent variable
 tf final value of independent variable
 u0 initial condition vector of length neqn
 nsteps number of rkf45 steps
 u ODE solution vector of length neqn after
 nsteps steps
 */
{
    /* Type variables */
    double h, t;
    double e[]=new double[SIZE];
    int i, j;

    /* Integration step */
    h=(tf-t0)/nsteps;

    /* nsteps rkf45 steps */
    for(i=1;i<=nsteps;i++)
    {
        /* Single rkf45 step */
        t=ssrkf45(neqn,t0,u0,h,u,e);

        /* Reset base point values for next rkf45 step */
        for(j=1;j<=neqn;j++)
        {
            u0[j]=u[j];
        }
        t0=t;
    }
Program 2.5.10
Integrator rkf45a

public void rkf45b(int neqn, double t0, double tf,
                   double u[], double u0[], int nsteps,
                   double abserr, double relerr)
{" Type variables "
  double hmin, h, t;
  double e[] = new double[SIZE];
  int i, nfin1;
"}
/* Integration step */
h=(tf-t0)/2.0;

/* Minimum allowable step */
hmin=(tf-t0)/nsteps;

/* Start integration */
t=t0;

/* While independent variable is less than the final value, continue the integration */
while(t <= (tf*0.999))
{
    /* If the next step along the solution will go past the final value of the independent variable, set the step to the remaining distance to the final value */
    if((t+h) > tf)
    {
        h=tf-t;
    }

    /* Single rkf45 step */
t=ssrkf45(neqn,t0,u0,h,u,e);

    /* Flag for the end of the integration */
nfinl=1;

    /* Check if any of the ODEs have violated the error criterion */
    for(i=1;i<=neqn;i++)
    {
        if(Math.abs(e[i]) > (Math.abs(u[i])*relerr+abserr))
        {
            /* Error violation, so integration is incomplete. Reduce integration step because of error violation and repeat integration from base point */
            nfinl=0;
            h=h/2.0;
            /* If the current step is less than the minimum allowable step, set the step to the minimum allowable value and continue integration from new base point */
if(h < hmin)
{
    h = hmin;
    nfin1=1;
}
break;
}

/* If there is no error violation, continue the integration from the new base point */
if(nfin1 == 1)
{
    for(i=1;i<=neqn;i++)
    {
        u0[i]=u[i];
    }
    t0=t;

    /* Test if integration step can be increased */
    for(i=1;i<=neqn;i++)
    {
        if(Math.abs(e[i]) > ((Math.abs(u[i]) *relerr+abserr)/32.0) )
        {
            /* Integration step cannot be increased */
            nfin1=0;
            break;
        }
    }

    /* Increase integration step */
    if(nfin1 == 1)
    {
        h=h*2.0;
    }

    /* End if */
}

/* End while */
Program 2.5.11
Integrator rkf45b

public double ssrkf45(int neqn, double t0, double u0[],
                       double h, double u[], double e[])
/*
 Function ssrkf45 computes an ODE solution by the RK
 Fehlberg 45 method for one step along the solution (by
calls to derv to define the ODE derivative vector). It
also estimates the truncation error of the solution, and
applies this estimate as a correction to the solution
vector.

Argument list

 neqn number of first order ODEs
 t0 initial value of independent variable
 u0 initial condition vector of length neqn
 h integration step
 t independent variable
 u ODE solution vector of length neqn after
 one rkf45 step
 e estimate of truncation error of the solu-
tion vector
*/
{
 /* Type variables */
 double t, ut0[], ut[], u5[];
 double k1[], k2[], k3[], k4[], k5[], k6[];
 int j;

 /* Declare arrays */
 ut0=new double[SIZE];
 ut=new double[SIZE];
 u5=new double[SIZE];
k1=new double[SIZE];
k2=new double[SIZE];
k3=new double[SIZE];
k4=new double[SIZE];
k5=new double[SIZE];
k6=new double[SIZE];

/* Derivative vector at initial (base) point */
derv(ut0,t0,u0);

/* k1; stepping for k2 */
for(j=1;j<=neqn;j++)
{
    k1[j]=h*ut0[j];
    u[j]=u0[j]+0.25*k1[j];
}
t=t0+0.25*h;

/* Derivative vector at next RK point */
derv(ut,t,u);

/* k2; stepping for k3 */
for(j=1;j<=neqn;j++)
{
    k2[j]=h*ut[j];
    u[j]=u0[j]+(3.0/32.0)*k1[j]
        +(9.0/32.0)*k2[j];
}
t=t0+(3.0/8.0)*h;

/* Derivative vector at next RK point */
derv(ut,t,u);

/* k3; stepping for k4 */
for(j=1;j<=neqn;j++)
{
    k3[j]=h*ut[j];
    u[j]=u0[j]+(1932.0/2197.0)*k1[j]
        -(7200.0/2197.0)*k2[j]
        +(7296.0/2197.0)*k3[j];
}
t=t0+(12.00/13.0)*h;

/* Derivative vector at next RK point */
derv(ut,t,u);
/* k4; stepping for k5 */
for(j=1;j<=neqn;j++)
{
    k4[j]=h*ut[j];
    u[j]=u0[j]+( 439.0 / 216.0)*k1[j]
    -( 8.0 )*k2[j]
    +(3680.0 / 513.0)*k3[j]
    -(845.0/4104.0)*k4[j];
}
t=t0+h;

/* Derivative vector at next RK point */
derv(ut,t,u);

/* k5; stepping for k6 */
for(j=1;j<=neqn;j++)
{
    k5[j]=h*ut[j];
    u[j]=u0[j]-( 8.0 / 27.0)*k1[j]
    +( 2.0 )*k2[j]
    -(3544.0/2565.0)*k3[j]
    +(1859.0/4104.0)*k4[j]
    -(11.0/ 40.0)*k5[j];
}
t=t0+0.5*h;

/* Derivative vector at next RK point */
derv(ut,t,u);

/* k6; fourth and fifth order step; error estimate; error correction */
for(j=1;j<=neqn;j++)
{
    k6[j]=h*ut[j];
    u[j]=u0[j]+( 25.0 / 216.0)*k1[j]
    +(1408.0/2565.0)*k3[j]
    +(2197.0/4104.0)*k4[j]
    -( 1.0/ 5.0)*k5[j];
    u5[j]=u0[j]+( 16.0/ 135.0)*k1[j]
    +(6656.0/12825.0)*k3[j]
    +(28561.0/56430.0)*k4[j]
    -( 9.0/ 50.0)*k5[j]
    +( 2.0/ 55.0)*k6[j];
    e[j]=u5[j]-u[j];
    u[j]=u[j]+e[j];
t=t0+h;
return t;

/* End of ssrkf45 */
}

/* End public class RK */
}

Program 2.5.12
Integrator ssrkf45 for a RKF45 step

Routines for the Runge Kutta ODE Integration

package mol;

import java.math.*;
import java.io.*;

public interface RKInterface
{
    /* Fixed step classical fourth order RK method */
    public void rkc4a(int neqn, double t0, double tf,
                     double u[], double u0[], int nsteps);

    /* Variable step classical fourth order RK method */
    public void rkc4b(int neqn, double t0, double tf,
                      double u[], double u0[], int nsteps,
                      double abserr, double relerr);

    /* Single step classical fourth order RK method */
    public double ssrkc4(int neqn, double t0, double u0[],
                         double h, double u[], double e[]);

    /* Fixed step RK Fehlberg 45 method */
    public void rkf45a(int neqn, double t0, double tf,
                       double u[], double u0[], int nsteps);
/* Variable step RK Fehlberg 45 method */
public void rkf45b(int neqn, double t0, double tf,
        double u[], double u0[], int nsteps,
        double abserr, double relerr);

/* Single step RK Fehlberg 45 method */
public double ssrkf45(int neqn, double t0, double u0[],
        double h, double u[], double e[]);

Program 2.5.13
Interface routine for rkc4a, rkc4b, ssrkc4, rkf45a, rkf45b, ssrkf45

intpar, initial, derv, and fprint follow:

/* This file is a member of the package mol */
package mol;

import mol.MOL;
import java.math.*;
import java.io.*;
import java.text.*;

public class DEF extends MOL implements ode1x1interface {
    public DEF() {
        /* Integration parameters */
        this.intpar();

        /* Declare arrays */
        u0=new double[SIZE];
        u=new double[SIZE];
        e=new double[SIZE];

        /* Initial condition vector */
        this.initial();

    }

    public void intpar() {
        /* Function intpar sets the parameters to control the
           integration of the 1 x 1 ODE system */
    }

Copyright 2004 by Chapman & Hall/CRC
/ * Number of ODEs */
neqn=1;

/* Size of arrays in MOL library */
SIZE=neqn+1;

/* Number of output points */
 nout=6;

/* Maximum number of steps in the interval t0 to tf */
 nsteps=100;

/* Initial, final values of the independent variable */
t0=0.0;
tf=1.0;

/* Error tolerances */
 abserr=Math.pow(10.0,-5.0);
 relerr=Math.pow(10.0,-5.0);

/* End of intpar */
}

public void initial()

/* Function initial sets the initial condition vector for
the 1 x 1 ODE problem */
{
 u0[1]=1.0E0;

/* End of initial */
}

public void deriv(double ut[], double t, double u[])

/* Function deriv computes the derivative vector of the
1 x 1 ODE problem */
{
 /* Type variables */
 double alpha, lambda;

 /* Problem parameters */
 alpha=1.0E0;
 lambda=1.0E0;
/ Derivative vector */
   ut[1]=lambda*Math.exp(-alpha*t)*u[1];

   /* End of derv */
}

public void fprint(PrintWriter f, int ncase, int neqn, double t, double u[])
/* Function fprint displays the numerical and exact solutions to the 1 x 1 ODE problem */
{
   /* Type variables */
   double ue1;
   double diff1;
   double e1;
   double V0, alpha, lambda;

   /* Problem parameters */
   V0=1.0E0;
   alpha=1.0E0;
   lambda=1.0E0;

   /* Print a heading for the solution at t = 0 */
   if(t<=0.0)
   {
      /* Label for ODE integrator */
      switch(ncase)
      {
      /*Fixed step modified Euler */
      case 1:
         f.println("\n euler2a integrator\n");
         break;

      /* Variable step modified Euler */
      case 2:
         f.println("\n euler2b integrator\n");
         break;

      /* Fixed step classical fourth order RK */
      case 3:
         f.println("\n rkc4a integrator\n");
         break;
      }
   }
}
/* Variable step classical fourth order RK */
case 4:
f.println("\n\n rkc4b integrator\n");
break;

/* Fixed step RK Fehlberg 45 */
case 5:
f.println("\n\n rkf45a integrator\n");
break;

/* Variable step RK Fehlberg 45 */
case 6:
f.println("\n\n rkf45b integrator\n");
break;

/* Heading */
f.println(" t u1(num) ue1 diff1");
/* End of t = 0 heading */
}

/* Analytical solution */
ue1=V0*Math.exp(lambda/alpha*(1.0E0-Math.exp(-alpha*t)));

/* Difference between exact and numerical solution vectors */
diff1=u[1]-ue1;

/* Display format for floating numbers */
DecimalFormat df1 = new DecimalFormat(" 0.00");
DecimalFormat df2 = new DecimalFormat("0.0000000");

/* Display the numerical and exact solutions, and their difference */
f.println(df1.format(t)+"\t"+df2.format(u[1])
          +"\t"+df2.format(ue1)+"\t"+df2.format(diff1));

/* End of fprintf */
}

/* End of DEF */
}
Program 2.5.14

\textit{intpar, initial, derv, and fprint} called in the solution of Equations 1.3 and 1.4

The output from the preceding routines (written to file \textit{ode1x1jv.out} in Program 2.5.1) is as follows:

\textbf{euler2a integrator}

\begin{tabular}{llll}
 t & u1(num) & uel & diff1 \\
0.00 & 1.0000000 & 1.0000000 & 0.0000000 \\
1.00 & 1.8815963 & 1.8815964 & -0.0000001 \\
2.00 & 2.3742138 & 2.3742099 & 0.0000039 \\
3.00 & 2.5862663 & 2.5862603 & 0.0000060 \\
4.00 & 2.6689549 & 2.6689479 & 0.0000069 \\
5.00 & 2.7000350 & 2.7000278 & 0.0000073 \\
\end{tabular}

\textbf{euler2b integrator}

\begin{tabular}{llll}
 t & u1(num) & uel & diff1 \\
0.00 & 1.0000000 & 1.0000000 & 0.0000000 \\
1.00 & 1.8815902 & 1.8815964 & -0.0000062 \\
2.00 & 2.3742061 & 2.3742099 & 0.0000038 \\
3.00 & 2.5862606 & 2.5862603 & 0.0000037 \\
4.00 & 2.6689516 & 2.6689479 & 0.0000037 \\
5.00 & 2.7000340 & 2.7000278 & 0.0000062 \\
\end{tabular}

\textbf{rkc4a integrator}

\begin{tabular}{llll}
 t & u1(num) & uel & diff1 \\
0.00 & 1.0000000 & 1.0000000 & 0.0000000 \\
1.00 & 1.8815964 & 1.8815964 & 0.0000000 \\
2.00 & 2.3742099 & 2.3742099 & 0.0000000 \\
3.00 & 2.5862603 & 2.5862603 & 0.0000000 \\
4.00 & 2.6689479 & 2.6689479 & 0.0000000 \\
5.00 & 2.7000278 & 2.7000278 & 0.0000000 \\
\end{tabular}

\textbf{rkc4b integrator}

\begin{tabular}{llll}
 t & u1(num) & uel & diff1 \\
0.00 & 1.0000000 & 1.0000000 & 0.0000000 \\
1.00 & 1.8815963 & 1.8815964 & 0.0000000 \\
2.00 & 2.3742100 & 2.3742099 & 0.0000001 \\
\end{tabular}
3.00  2.5862604  2.5862603  0.0000001  
4.00  2.6689480  2.6689479  0.0000001  
5.00  2.7000279  2.7000278  0.0000001  

rkf45a integrator

<table>
<thead>
<tr>
<th>t</th>
<th>u1(num)</th>
<th>ue1</th>
<th>diff1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.0000000</td>
<td>1.0000000</td>
<td>0.0000000</td>
</tr>
<tr>
<td>1.00</td>
<td>1.8815964</td>
<td>1.8815964</td>
<td>0.0000000</td>
</tr>
<tr>
<td>2.00</td>
<td>2.3742099</td>
<td>2.3742099</td>
<td>0.0000000</td>
</tr>
<tr>
<td>3.00</td>
<td>2.5862603</td>
<td>2.5862603</td>
<td>0.0000000</td>
</tr>
<tr>
<td>4.00</td>
<td>2.6689479</td>
<td>2.6689479</td>
<td>0.0000000</td>
</tr>
<tr>
<td>5.00</td>
<td>2.7000278</td>
<td>2.7000278</td>
<td>0.0000000</td>
</tr>
</tbody>
</table>

rkf45b integrator

<table>
<thead>
<tr>
<th>t</th>
<th>u1(num)</th>
<th>ue1</th>
<th>diff1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.0000000</td>
<td>1.0000000</td>
<td>0.0000000</td>
</tr>
<tr>
<td>1.00</td>
<td>1.8815931</td>
<td>1.8815964</td>
<td>-0.0000033</td>
</tr>
<tr>
<td>2.00</td>
<td>2.3742053</td>
<td>2.3742099</td>
<td>-0.0000046</td>
</tr>
<tr>
<td>3.00</td>
<td>2.5862553</td>
<td>2.5862603</td>
<td>-0.0000050</td>
</tr>
<tr>
<td>4.00</td>
<td>2.6689428</td>
<td>2.6689479</td>
<td>-0.0000052</td>
</tr>
<tr>
<td>5.00</td>
<td>2.7000225</td>
<td>2.7000278</td>
<td>-0.0000052</td>
</tr>
</tbody>
</table>

We note again that the output verifies the operation of all six integrators. The fixed step integrators (rkc4a, rkf45a, and ssrkc4) generally surpass the error tolerances set in intpar, with rkf45a and ssrkc4 producing excessive accuracy; the variable step integrators (rkc4b, rkf45b, and ssrkc4) produce numerical solutions consistent with the error tolerances.

This completes the programming of the 1x1 ODE problem in Java. We now move on to Maple.

2.6 Programming in Maple

We conclude the programming of the 1x1 ODE problem using Maple. Also, we have used the following approach, which is a small departure from the preceding programming and the more typical programming in Maple:

- Maple files, with the extension \textit{.mws} (for \textit{Maple Worksheet}), are ASCII files, but they cannot be read using, for example, an ASCII text editor; rather they must be viewed under Maple. We, therefore, have used Maple coded
as readable ASCII files, and have given the resulting files the extension .txt. Then these .txt files are opened within the Maple code (via read statements). This may seem complicated, but the following coding will demonstrate that this procedure is straightforward. This approach has the major advantage that the Maple code can be written outside of Maple (in fact, we used the MATLAB routines in Section 2.1, which are readable ASCII, as templates for writing the corresponding Maple).

- Since the .txt files are opened using the Maple read statement, the separation between general-purpose (library) routines and the application-specific routines cannot be maintained as well as before. In other words, the read statements in the general-purpose routines have to be modified to open specific applications files. This is a disadvantage, but we view it as minor compared with the advantage of using readable ASCII (.txt) files.

The details of this approach will become clear through the following coding for the 1x1 ODE problem. We start with a small (three line).mws main program that in turn executes a series of read statements for the .txt files:

```maple
> restart:
> read "c:\odelib\maple\ode1x1\ode1x1.txt";
> ode1x1();
```

**Program 2.6.1**

Maple main program ode1x1.mws for the numerical integration of Equation 1.3 subject to initial condition Equation 1.4

We can note the following points about this main program:

- The main program ode1x1.txt is read (which means that it is now available for execution):
  ```maple
  > read "c:\odelib\maple\ode1x1\ode1x1.txt";
  
  ode1x1, which is a Maple procedure, is then executed:
  ```maple
  > ode1x1();
  ```

Procedure ode1x1 is listed below:

```maple
ode1x1:=proc()
  
  # Main program ode1x1 computes the numerical
  # solution to the 1 x 1 ODE system by one of
  # six integrators
```
# Type variables
global neqn, nout, nsteps, t0, tf, abserr, relerr:
local u0, u, tp, ncase, i, j:
#
# Step through six integrators
for ncase from 1 to 6 do
#
# Integration parameters
read "c:\odelib\maple\ode1x1\intpar.txt":
intpar():
#
# Size arrays
u0:=array(1..neqn): u:=array(1..neqn):
#
# Initial condition vector
read "c:\odelib\maple\ode1x1\inital.txt":
inital(n,t0,u0):
#
# Output interval
tp:=tf-t0:
#
# Compute solution at nout output points
for j from 1 to nout do
#
# Print current solution
read "c:\odelib\maple\ode1x1\fprint.txt":
fprint(ncase,neqn,t0,u0):
#
# Fixed step modified Euler integrator
if (ncase = 1) then
  read "c:\odelib\maple\ode2x2\euler2a.txt":
euler2a(neqn,t0,tf,u0,nsteps,u):
end if:
#
# Variable step modified Euler integrator
if (ncase = 2) then
  read "c:\odelib\maple\ode2x2\euler2b.txt":
euler2b(neqn,t0,tf,u0,nsteps,abserr,relerr,u):
end if:
#
# Fixed step classical fourth order RK integrator
if (ncase = 3) then
  read "c:\odelib\maple\ode2x2\rkc4a.txt":
rkc4a(neqn,t0,tf,u0,nsteps,u):
end if:

# Variable step classical fourth order RK integrator
if (ncase = 4) then
    read "c:\odelib\maple\ode2x2\rkc4b.txt":
    rkc4b(neqn,t0,tf,u0,nsteps,abserr,relerr,u):
end if:
#
# Fixed step RK Fehlberg (RKF45) integrator
if (ncase = 5) then
    read "c:\odelib\maple\ode2x2\rkf45a.txt":
    rkf45a(neqn,t0,tf,u0,nsteps,u):
end if:
#
# Variable step RK Fehlberg (RKF45) integrator
if (ncase = 6) then
    read "c:\odelib\maple\ode2x2\rkf45b.txt":
    rkf45b(neqn,t0,tf,u0,nsteps,abserr,relerr,u):
end if:
#
# Advance solution
    t0:=tf:
    tf:=tf+tp:
    for i from 1 to neqn do
        u0[i]:=u[i]:
    end do:
#
# Next output
    end do:
#
# Next integrator
    end do:
#
# End of ode1x1.txt
    end:

Program 2.6.2
Maple main program ode1x1.txt for the numerical integration of Equation 1.3
subject to initial condition Equation 1.4

ode1x1.txt has a familiar structure. We can note the following points:

- # denotes a comment in Maple.
- Again, all six integrators are called within a single loop:
# Step through six integrators
for ncase from 1 to 6 do

• Within this loop, each of the ODE integrators, euler2a, euler2b, sseuler, rkc4a, rkc4b, ssrkc4, rkf45a, rkf45b, and ssrkf45, is opened via a read statement.

• Additionally, routines intpar, initial, and fprint, which are specific to the 1x1 ODE application, are opened via read statements; thus, the separation of general-purpose and applications-specific routines is not as clear-cut as in the discussion of the preceding five languages.

The ODE integration routines are listed below. Generally they have the same form as before; the file read statements are the only main difference.

euler2a:=proc(neqn,t0,tf,u0,nsteps,u)
#
# Procedure euler2a computes an ODE solution by a fixed
# step modified Euler method for a series of points along
# the solution by repeatedly calling procedure sseuler
# for a single modified Euler step.
#
# Argument list
#
# neqn number of first order ODEs
#
# t 0 initial value of independent variable
#
# tf final value of independent variable
#
# u 0 initial condition vector of length neqn
#
# nsteps number of modified Euler steps
#
# u ODE solution vector of length neqn after
# nsteps steps
#
# Type variables
local e, h, i, j, tm, t:
#
# Size arrays
e:=array(1..neqn):
#
# Integration step
h:=(tf-t0)/nsteps:
# nsteps modified Euler steps
tm:=t0:
for j from 1 to nsteps do
  # Modified Euler step
  read "c:\odelib\maple\ode2x2\sseuler.txt":
  sseuler(neqn,tm,u0,h,u,e):
  t:=tm+h:
  # Reset base point values for next modified Euler step
  for i from 1 to neqn do
    u0[i]:=u[i]:
  end do:
  tm:=t:
end do:
# Next modified Euler step
end:
# End of euler2a
end:

Program 2.6.3
Integrator euler2a

euler2b:=proc(neqn,t0,tf,u0,nsteps,abserr,relerr,u)
  # Procedure euler2b computes an ODE solution by a variable
  # step modified Euler method for a series of points along
  # the solution by repeatedly calling procedure sseuler for
  # a single modified Euler step. The truncation error is
  # estimated along the solution to adjust the integration
  # step according to a specified error tolerance.
  # Argument list
  #
  # neqn number of first order ODEs
  # t0 initial value of independent variable
  # tf final value of independent variable
  # u0 initial condition vector of length neqn
  # nsteps maximum number of modified Euler steps

# abserr absolute error tolerance
# relerr relative error tolerance
# u ODE solution vector of length eqn after
# nsteps steps
# Type variables
local e, h, i, j, tm, t, hmin, nfin1:
#
# Size arrays
e:=array(1..neqn):
#
# Initial integration step
h:=(tf-t0)/8.0:
#
# Minimum allowable step
hmin:=(tf-t0)/nsteps:
#
# Start integration
tm:=t0:
#
# While independent variable is less than the final
# value, continue the integration
while(tm <= tf*0.999) do
#
# If the next step along the solution will go past
# the final value of the independent variable, set
# the step to the remaining distance to the final
# value
if(tm+h > tf) then h:=tf-tm: end if:
#
# Single modified Euler step
read "c:\odelib\maple\ode2x2\sseuler.txt":
sseuler(neqn,tm,u0,h,u,e):
t:=tm+h:
#
# Flag for the end of the integration
nfin1:=1:
#
# Check if any of the ODEs have violated the error
# criteria
for i from 1 to neqn do
if(abs(e[i]) > abs(u[i])*relerr+abserr) then
# Error violation, so integration is not complete.
# Reduce integration step because of error violation
# and repeat integration from the base point
  h := h/2.0; nfin1 := 0; break;
end if:
end do:

# If the current step is less than the minimum
# allowable step, set the step to the minimum
# allowable value and continue integration from
# new base point
  if (h < hmin) then h := hmin; nfin1 := 1; end if:

# If there is no error violation, continue the
# integration from new base point
  if (nfin1 = 1) then
    for i from 1 to neqn do
      u0[i] := u[i];
    end do;
    tm := t:
  end if:

# Test if integration step can be increased
  for i from 1 to neqn do
    if (abs(e[i]) > (abs(u[i])*relerr+abserr)/4.0) then
      Integration step cannot be increased
      nfin1 := 0; break:
    end if:
  end do:

# Increase integration step
  if (nfin1 = 1) then h := h*2.0; end if:

# Continue for no error violation (nfin1=1)
  end if:

# Continue while
end do:

# End of euler2b
end:

Program 2.6.4
Integrator euler2b
sseuler:=proc(neqn,t0,u0,h,u,e)
#
# Procedure sseuler computes an ODE solution by the
# modified Euler method for one step along the solution
# (by calls to derv to define the ODE derivative vector).
# It also estimates the truncation error of the solution,
# and applies this estimate as a correction to the
# solution vector.
#
# Argument list
#
# neqn number of first order ODEs
#
# t0 initial value of independent variable
#
# u0 initial condition vector of length neqn
#
# h integration step
#
# u ODE solution vector of length neqn after
# one modified Euler step
#
# e estimate of truncation error of the solution vector
#
# Type variables
local ut0, ut, t, i:
#
# Declare arrays
ut0:=array(1..neqn): ut:=array(1..neqn):
#
# Derivative vector at initial (base) point
read "c:\\oedelib\\maple\\odeix1\\derv.txt":
derv(neqn,t0,u0,ut0):
#
# First order (Euler) step
for i from 1 to neqn do
u[i]:=u0[i]+ut0[i]*h:
end do:
t:=t0+h:
#
# Derivative vector at advance point
derv(neqn,t,u,ut):
#
# Truncation error estimate
for i from 1 to neqn do
    e[i] := (ut[i] - ut0[i]) * h/2.0;
end do;
#
# Second order (modified Euler) solution vector
for i from 1 to neqn do
    u[i] := u[i] + e[i];
end do;
#
# End of sseuler
end:

Program 2.6.5
Integrator sseuler for a single modified Euler step

Note in sseuler the read statement to the application specific derv:
#
# Derivative vector at initial (base) point
read "c:\odelib\maple\ode1x1\derv.txt":
derv(neqn, t0, u0, ut0):

rkc4a := proc(neqn, t0, tf, u0, nsteps, u)
#
# Procedure rkc4a computes an ODE solution by a fixed step
# classical fourth order RK method for a series of points
# along the solution by repeatedly calling procedure ssrkc4
# for a single classical fourth order RK step.
#
# Argument list
#
# neqn number of first order ODEs
# t0 initial value of independent variable
# tf final value of independent variable
# u0 initial condition vector of length neqn
# nsteps number of rkc4 steps
# u ODE solution vector of length neqn after
# nsteps steps
#
# Type variables
local e, h, i, j, tm, t:
# Size arrays
e:=array(1..neqn):
#
# Integration step
h:=(tf-t0)/nsteps:
#
# nsteps rkc4 steps
tm:=t0:
for j from 1 to nsteps do
#
# rkc4 step
read "c:\odelib\maple\ode2x2\ssrkc4.txt":
ssrkc4(neqn,tm,u0,h,u,e):
t:=tm+h:
#
# Reset base point values for next rkc4 step
for i from 1 to neqn do
   u0[i]:=u[i]:
end do:
   tm:=t:
#
# Next rkc4 step
end do:
#
# End of rkc4a
end:

Program 2.6.6
Integrator rkc4a

rkc4b:=proc(neqn,t0,tf,u0,nsteps,abserr,relerr,u)
#
# Procedure rkc4b computes an ODE solution by a variable
# step classical fourth order RK method for a series of
# points along the solution by repeatedly calling procedure
# ssrkc4 for a single classical fourth order RK step. The
# truncation error is estimated along the solution to
# adjust the integration step according to a specified
# error tolerance.
#
# Argument list
#
# neqn number of first order ODEs
# t0 initial value of independent variable
# tf final value of independent variable
# u0 initial condition vector of length neqn
# nsteps maximum number of rkc4 steps
# abserr absolute error tolerance
# relerr relative error tolerance
# u ODE solution vector of length neqn after
# nsteps steps
#
# Type variables
local e, h, i, j, tm, t, hmin, nfin1:
#
# Size arrays
e:=array(1..neqn):
#
# Initial integration step
h:=(tf-t0)/2.0:
#
# Minimum allowable step
hmin:=(tf-t0)/nsteps:
#
# Start integration
tm:=t0:
#
# While independent variable is less than the final
# value, continue the integration
while(tm <= tf*0.999) do
#
# If the next step along the solution will go past
# the final value of the independent variable, set
# the step to the remaining distance to the final
# value
if(tm+h > tf) then h:=tf-tm: end if:
#
# Single rkc4 step
read "c:\odelib\maple\ode2x2\ssrkc4.txt":
ssrkc4(neqn,tm,u0,h,u,e):
t:=tm+h:
# Flag for the end of the integration
nfin1:=1:
#
# Check if any of the ODEs have violated the error
# criteria
for i from 1 to neqn do
    if(abs(e[i]) > abs(u[i])*relerr+abserr) then
        # Error violation, so integration is not complete.
        # Reduce integration step because of error violation
        # and repeat integration from the base point
        h:=h/2.0: nfin1:=0: break:
    end if:
end do:
#
# If the current step is less than the minimum
# allowable step, set the step to the minimum
# allowable value and continue integration from
# new base point
if(h < hmin) then h:=hmin: nfin1:=1: end if:
#
# If there is no error violation, continue the
# integration from new base point
if(nfin1 = 1) then
    for i from 1 to neqn do
        u0[i]:=u[i]:
    end do:
    tm:=t:
#
# Test if integration step can be increased
for i from 1 to neqn do
    if(abs(e[i]) > (abs(u[i])*relerr+abserr)/16.0) then
        # Integration step cannot be increased
        nfin1:=0: break:
    end if:
end do:
#
# Increase integration step
if(nfin1 = 1) then h:=h*2.0: end if:
#
# Continue for no error violation (nfin1=1)
end if:
ssrk4:=proc(neqn,t0,u0,h,u,e)
# Procedure ssrk4 computes an ODE solution by the
# classical fourth order RK method for one step along
# the solution (by calls to derv to define the ODE
# derivative vector). It also estimates the truncation
# error of the solution, and applies this estimate as a
# correction to the solution vector.
#
# Argument list
#
# neqn number of first order ODEs
#
# t0 initial value of independent variable
#
# u0 initial condition vector of length neqn
#
# h integration step
#
# u ODE solution vector of length neqn after
# one modified Euler step
#
# e estimate of truncation error of the solution vector
#
# Type variables
local ut0, ut, t, i, k1, k2, k3, k4, sum2, sum4:
#
# Declare arrays
ut0:=array(1..neqn): ut:=array(1..neqn):
  k1:=array(1..neqn): k2:=array(1..neqn):
  k3:=array(1..neqn): k4:=array(1..neqn):
  sum2:=array(1..neqn): sum4:=array(1..neqn):
#
# Derivative vector at initial (base) point
read "c:\odelib\maple\odex1\derv.txt":
derv(neqn,t0,u0,ut0):
#
# k1, advance of dependent variable vector and
# independent variable for calculation of k2
for i from 1 to neqn do
    k1[i]:=h*ut0[i]:
    u[i]:=u0[i]+0.5*k1[i]:
end do:

# Derivative vector at new u, t
derv(neqn,t,u,ut):
#
# k2, advance of dependent variable vector and
# independent variable for calculation of k3
for i from 1 to neqn do
    k2[i]:=h*ut[i]:
    u[i]:=u0[i]+0.5*k2[i]:
end do:

# Derivative vector at new u, t
derv(neqn,t,u,ut):
#
# k3, advance of dependent variable vector and
# independent variable for calculation of k4
for i from 1 to neqn do
    k3[i]:=h*ut[i]:
    u[i]:=u0[i]+k3[i]:
end do:

# Derivative vector at new u, t
derv(neqn,t,u,ut):
#
# k4
for i from 1 to neqn do
    k4[i]:=h*ut[i]:
end do:

# Second order step
for i from 1 to neqn do
    sum2[i]:=u0[i]+k2[i]:
end do:
#
# Fourth order step
for i from 1 to neqn do
    sum4[i]:=u0[i]+(1.0/6.0)*(k1[i]+2.0*k2[i]+2.0*k3[i]+k4[i])
end do:
t:=t0+h:
#
# Truncation error estimate
for i from 1 to neqn do
    e[i]:=sum4[i]-sum2[i]:
end do:
#
# Fourth order solution vector (from 2,4 RK pair);
# two ways to the same result are listed
for i from 1 to neqn do
    u[i]:=sum2[i]+e[i]:
    u[i]:=sum4[i]:
end do:
#
# End of ssrk4
end:

Program 2.6.8
Integrator ssrk4 for a classical fourth-order RK step

Again, in ssrk4 the read statement is for the application specific deriv:

# Derivative vector at initial (base) point
read "c:\odelib\maple\ode1x1\der.txt":
deriv(neqn,t0,u0,ut0):
rkf45a:=proc(neqn,t0,tf,u0,nsteps,u)
    # Procedure rkf45a computes an ODE solution by the fixed
    # step RK Fehlberg 45 method for a series of points along
    # the solution by repeatedly calling procedure ssrkf45 for
    # a single RK Fehlberg 45 step.
    # Argument list
    # neqn number of first order ODEs
# t0       initial value of independent variable
#
# tf       final value of independent variable
#
# u0       initial condition vector of length neqn
#
# nsteps   number of rkf45 steps
#
# u        ODE solution vector of length neqn after
#          nsteps steps
#
# Type variables
local e, h, i, j, tm, t:
#
# Size arrays
e:=array(1..neqn):
#
# Integration step
h:=(tf-t0)/nsteps:
#
# nsteps rkf45 steps
 tm:=t0:
for j from 1 to nsteps do

  # rkf45 step
  read "c:\odelib\maple\ode2x2\ssrkf45.txt":
  ssrkf45(neqn,tm,u0,h,u,e):
  t:=tm+h:

  # Reset base point values for next rkf45 step
  for i from 1 to neqn do
    u0[i]:=u[i]:
  end do:
  tm:=t:

  # Next rkf45 step
  end do:

# End of rkf45a
end:

Program 2.6.9
Integrator rkf45a
rkf45b:=proc(neqn,t0,tf,u0,nsteps,abserr,relerr,u)
# Procedure rkf45b computes an ODE solution by the variable
# step RK Fehlberg 45 method for a series of points along
# the solution by repeatedly calling procedure ssrkf45 for
# a single RK Fehlberg 45 step. The truncation error is
# estimated along the solution to adjust the integration
# step according to a specified error tolerance.
#
# Argument list
#
# neqn number of first order ODEs
# t0 initial value of independent variable
# tf final value of independent variable
# u0 initial condition vector of length neqn
# nsteps maximum number number of rkf45 steps
# abserr absolute error tolerance
# relerr relative error tolerance
# u ODE solution vector of length neqn after
# nsteps steps
#
# Type variables
local e, h, i, j, tm, t, hmin, nfin1:
#
# Size arrays
e:=array(1..neqn):
#
# Initial integration step
h:=(tf-t0)/2.0:
#
# Minimum allowable step
hmin:=(tf-t0)/nsteps:
#
# Start integration
tm:=t0:
#
# While independent variable is less than the final
# value, continue the integration
while(tm <= tf*0.999) do
# If the next step along the solution will go past
# the final value of the independent variable, set
# the step to the remaining distance to the final
# value
  if(tm+h > tf) then h:=tf-tm: end if:
#
# Single rkf45 step
  read "c:\odelib\maple\ode2x2\ssrkf45.txt":
  ssrkf45(neqn,tm,u0,h,u,e):
  t:=tm+h:
#
# Flag for the end of the integration
  nfin1:=1:
#
# Check if any of the ODEs have violated the error
# criteria
  for i from 1 to neqn do
    if(abs(e[i]) > abs(u[i])*relerr+abserr) then
      # Error violation, so integration is not complete.
      # Reduce integration step because of error violation
      # and repeat integration from the base point
      h:=h/2.0: nfin1:=0: break:
    end if:
  end do:
#
# If the current step is less than the minimum
# allowable step, set the step to the minimum
# allowable value and continue integration from
# new base point
  if(h < hmin) then h:=hmin: nfin1:=1: end if:
#
# If there is no error violation, continue the
# integration from new base point
  if(nfin1 = 1) then
    for i from 1 to neqn do
      u0[i]:=u[i]:
    end do:
    tm:=t:
  #
  # Test if integration step can be increased
  for i from 1 to neqn do
    if(abs(e[i]) > (abs(u[i])*relerr+abserr)/32.0) then
      #
Integration step cannot be increased
    nfin1:=0: break:
    end if:
end do:

Increase integration step
    if(nfin1 = 1) then h:=h*2.0: end if:

Continue for no error violation (nfin1=1)
    end if:

Continue while
end do:

End of rkf45b
end:

Program 2.6.10
Integrator rkf45b

ssrkf45:=proc(neqn,t0,u0,h,u,e)

# Procedure ssrkf45 computes an ODE solution by the RK Fehlberg 45 method for one step along the solution (by calls to derv to define the ODE derivative vector). It also estimates the truncation error of the solution, and applies this estimate as a correction to the solution vector.

# Argument list
# neqn number of first order ODEs
# t0 initial value of independent variable
# u0 initial condition vector of length neqn
# h integration step
# t independent variable
# u ODE solution vector of length neqn after one rkf45 step
# estimate of truncation error of the solution vector

# Type variables
local ut0, ut, t, i, k1, k2, k3, k4, k5, k6, sum4, sum5:

# Declare arrays
ut0:=array(1..neqn): ut:=array(1..neqn):
k1:=array(1..neqn):  k2:=array(1..neqn):
k3:=array(1..neqn):  k4:=array(1..neqn):
k5:=array(1..neqn):  k6:=array(1..neqn):
sum4:=array(1..neqn): sum5:=array(1..neqn):

# Derivative vector at initial (base) point
read "c:\odelib\maple\odeix1\derv.txt":
derv(neqn,t0,u0,ut0):

# k1, advance of dependent variable vector and independent variable for calculation of k2
for i from 1 to neqn do
  k1[i]:=h*ut0[i]:
  u[i]:=u0[i]+0.25*k1[i]:
end do:
t:=t0+0.25*h:

# Derivative vector at new u, t
derv(neqn,t,u,ut):

# k2, advance of dependent variable vector and independent variable for calculation of k3
for i from 1 to neqn do
  k2[i]:=h*ut[i]:
  u[i]:=u0[i]+(3.0/32.0)*k1[i]+(9.0/32.0)*k2[i]:
end do:
t:=t0+(3.0/8.0)*h:

# Derivative vector at new u, t
derv(neqn,t,u,ut):

# k3, advance of dependent variable vector and independent variable for calculation of k4
for i from 1 to neqn do
  k3[i]:=h*ut[i]:
\[
\begin{align*}
    u[i] &= u0[i] + \left(1932.0/2197.0\right) * k1[i] \\
    &\quad -\left(7200.0/2197.0\right) * k2[i] \\
    &\quad +\left(7296.0/2197.0\right) * k3[i] \\
    \end{align*}
\]

\text{end do:}
\[
\begin{align*}
    t &= t0 + \left(12.00/13.0\right) * h \\
\end{align*}
\]

\text{# Derivative vector at new u, t}
\text{derv(neqn,t,u,ut):}

\text{# k4, advance of dependent variable vector and}
\text{# independent variable for calculation of k5}
\text{for i from 1 to neqn do}
\text{k4[i] := \text{h} * ut[i]:}
\text{u[i] := u0[i] + \left(439.0/216.0\right) * k1[i] \\
    &\quad -\left(8.0\right) * k2[i] \\
    &\quad +\left(3680.0/513.0\right) * k3[i] \\
    &\quad -\left(845.0/4104.0\right) * k4[i]:}
\text{end do:}
\text{t := t0 + h:}

\text{# Derivative vector at new u, t}
\text{derv(neqn,t,u,ut):}

\text{# k5, advance of dependent variable vector and}
\text{# independent variable for calculation of k6}
\text{for i from 1 to neqn do}
\text{k5[i] := \text{h} * ut[i]:}
\text{u[i] := u0[i] - \left(8.0/27.0\right) * k1[i] \\
    &\quad +\left(2.0\right) * k2[i] \\
    &\quad -\left(3544.0/2565.0\right) * k3[i] \\
    &\quad +\left(1859.0/4104.0\right) * k4[i] \\
    &\quad -\left(11.0/40.0\right) * k5[i]:}
\text{end do:}
\text{t := t0 + 0.5 * h:}

\text{# Derivative vector at new u, t}
\text{derv(neqn,t,u,ut):}

\text{# k6}
\text{for i from 1 to neqn do}
\text{k6[i] := \text{h} * ut[i]:}
\text{end do:}

\text{# Fourth order step}
\text{for i from 1 to neqn do}

sum4[i]:=u0[i]+( 25.0/ 216.0)*k1[i]  
+ (1408.0/2565.0)*k3[i]  
+ ( 2197.0/4104.0)*k4[i]  
- ( 1.0/ 5.0)*k5[i]:
end do:
#
# Fifth order step
for i from 1 to neqn do
  sum5[i]:=u0[i]+( 16.0/ 135.0)*k1[i]  
  +( 6656.0/12825.0)*k3[i]  
  +(28561.0/56430.0)*k4[i]  
  - ( 9.0/ 50.0)*k5[i] 
  +( 2.0/ 55.0)*k6[i]:
end do:
t:=t0+h;
#
# Truncation error estimate
for i from 1 to neqn do
  e[i]:=sum5[i]-sum4[i]:
end do:
#
# Fifth order solution vector (from 4,5 RK pair);
# two ways to the same result are listed
for i from 1 to neqn do
  # u[i]:=sum4[i]+e[i]:
  u[i]:=sum5[i]:
end do:
#
# End of ssrkf45
end:

Program 2.6.11
Integrator ssrkf45 for an RKF45 step

Finally, intpar, initial, deriv, and fprintf are listed next:

intpar:=proc()
#
# Function intpar sets the parameters to control the
# integration of the 1 x 1 ODE system
#
# Type variables
  global neqn, nout, nsteps, t0, tf, abserr, relerr:
#
# Number of first order ODEs
  neqn:=1:
# Number of output points
nout:=6:
#
# Maximum number of steps in the interval t0 to tf
nsteps:=100:
#
# Initial, final values of independent variable
t0:=0.0:
tf:=1.0:
#
# Error tolerances
abserr:=1.0e-05:
relerr:=1.0e-05:
#
# End of intpar
end:

initial:=proc(neqn,t,u0)
#
# Procedure initial sets the initial condition vector
# for the 1 x 1 ODE problem
#
# End of initial
end:

derv:=proc(neqn,t,u,ut)
#
# Procedure derv computes the derivative vector
# of the 1 x 1 ODE problem
#
# Type variables
local alpha, lambda:
#
# Problem parameters
alpha:=1.0:
lambda:=1.0:
#
# Derivative vector
ut[1]:=lambda*exp(-alpha*t)*u[1]:
#
fprint:=proc(ncase,neqn,t,u)

# Procedure fprint displays the numerical and
# exact solutions to the 1 x 1 ODE problem
#
# Type variables
local u0, alpha, lambda, ue, diff, i:
#
# Problem parameters
u0:=1.0:
alpha:=1.0:
lambda:=1.0:
#
# Define arrays
ue:=array(1..neqn): diff:=array(1..neqn):
#
# Print a heading for the solution at t = 0
if (t <= 0.0) then
#
# Label for ODE integrator
#
# Fixed step modified Euler
if (ncase = 1) then
printf(`
\n
 euler2a integrator

`);
#
# Variable step modified Euler
elif (ncase = 2) then
printf(`
\n\n euler2b integrator\n\n`);
#
# Fixed step classical fourth order RK
elif (ncase = 3) then
printf(`
\n\n rkc4a integrator\n\n`);
#
# Variable step classical fourth order RK
elif (ncase = 4) then
printf(`
\n\n rkc4b integrator\n\n`);
#
# Fixed step RK Fehlberg 45
elif (ncase = 5) then
printf(`
\n\n rkf45a integrator\n\n`);
#
# Variable step RK Fehlberg 45
elif (ncase = 6) then
    printf(`

 rkf45b integrator

`);
end if:
#
# Heading
printf(`

 t u1 u1e u1-u1e
`):
#
# End of t = 0 heading
end if:
#
# Numerical and analytical solution output
#
# Analytical solution
ue[1]:=u0*exp(lambda/alpha*(1.0-exp(-alpha*t))):
#
# Difference between exact and numerical solutions
for i from 1 to neqn do
diff[i]:=u[i]-ue[i]:
end do:
#
# Display the numerical and exact solutions, and their
difference
printf(`%10.2f %10.5f %10.5f %10.5f 
`,t,u[1],ue[1],diff[1]);
#
# End of fprint
end:

Program 2.6.12
intpar, initial, derv, and fprint called in the solution of Equations 1.3 and 1.4

The output from the preceding routines is as follows:

euler2a integrator

<table>
<thead>
<tr>
<th>t</th>
<th>u1</th>
<th>u1e</th>
<th>u1-u1e</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.00000</td>
<td>1.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>1.00</td>
<td>1.88160</td>
<td>1.88160</td>
<td>-0.00000</td>
</tr>
<tr>
<td>2.00</td>
<td>2.37421</td>
<td>2.37421</td>
<td>0.00000</td>
</tr>
<tr>
<td>3.00</td>
<td>2.58627</td>
<td>2.58626</td>
<td>0.00001</td>
</tr>
<tr>
<td>4.00</td>
<td>2.66895</td>
<td>2.66895</td>
<td>0.00001</td>
</tr>
<tr>
<td>5.00</td>
<td>2.70004</td>
<td>2.70003</td>
<td>0.00001</td>
</tr>
</tbody>
</table>
### euler2b integrator

<table>
<thead>
<tr>
<th>t</th>
<th>ul</th>
<th>ule</th>
<th>ul-ule</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.00000</td>
<td>1.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>1.00</td>
<td>1.88159</td>
<td>1.88160</td>
<td>-.00001</td>
</tr>
<tr>
<td>2.00</td>
<td>2.37421</td>
<td>2.37421</td>
<td>-.00000</td>
</tr>
<tr>
<td>3.00</td>
<td>2.58626</td>
<td>2.58626</td>
<td>.00000</td>
</tr>
<tr>
<td>4.00</td>
<td>2.66895</td>
<td>2.66895</td>
<td>.00000</td>
</tr>
<tr>
<td>5.00</td>
<td>2.70003</td>
<td>2.70003</td>
<td>.00001</td>
</tr>
</tbody>
</table>

### rkc4a integrator

<table>
<thead>
<tr>
<th>t</th>
<th>ul</th>
<th>ule</th>
<th>ul-ule</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.00000</td>
<td>1.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>1.00</td>
<td>1.88160</td>
<td>1.88160</td>
<td>.00000</td>
</tr>
<tr>
<td>2.00</td>
<td>2.37421</td>
<td>2.37421</td>
<td>.00000</td>
</tr>
<tr>
<td>3.00</td>
<td>2.58626</td>
<td>2.58626</td>
<td>.00000</td>
</tr>
<tr>
<td>4.00</td>
<td>2.66895</td>
<td>2.66895</td>
<td>.00000</td>
</tr>
<tr>
<td>5.00</td>
<td>2.70003</td>
<td>2.70003</td>
<td>.00000</td>
</tr>
</tbody>
</table>

### rkc4b integrator

<table>
<thead>
<tr>
<th>t</th>
<th>ul</th>
<th>ule</th>
<th>ul-ule</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.00000</td>
<td>1.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>1.00</td>
<td>1.88160</td>
<td>1.88160</td>
<td>-.00000</td>
</tr>
<tr>
<td>2.00</td>
<td>2.37421</td>
<td>2.37421</td>
<td>-.00000</td>
</tr>
<tr>
<td>3.00</td>
<td>2.58626</td>
<td>2.58626</td>
<td>-.00000</td>
</tr>
<tr>
<td>4.00</td>
<td>2.66895</td>
<td>2.66895</td>
<td>-.00000</td>
</tr>
<tr>
<td>5.00</td>
<td>2.70003</td>
<td>2.70003</td>
<td>.00000</td>
</tr>
</tbody>
</table>

### rkf45a integrator

<table>
<thead>
<tr>
<th>t</th>
<th>ul</th>
<th>ule</th>
<th>ul-ule</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.00000</td>
<td>1.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>1.00</td>
<td>1.88160</td>
<td>1.88160</td>
<td>-.00000</td>
</tr>
<tr>
<td>2.00</td>
<td>2.37421</td>
<td>2.37421</td>
<td>-.00000</td>
</tr>
<tr>
<td>3.00</td>
<td>2.58626</td>
<td>2.58626</td>
<td>-.00000</td>
</tr>
<tr>
<td>4.00</td>
<td>2.66895</td>
<td>2.66895</td>
<td>-.00000</td>
</tr>
<tr>
<td>5.00</td>
<td>2.70003</td>
<td>2.70003</td>
<td>-.00000</td>
</tr>
</tbody>
</table>
This output is similar to that considered previously, e.g., the error tolerances set in `intpar` have been observed.

This concludes the programming of the 1x1 ODE problem in the six languages. We have discussed general-purpose routines for the solution of systems of ODEs in the following sections of Chapter 2:

<table>
<thead>
<tr>
<th>Section</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>MATLAB</td>
</tr>
<tr>
<td>2.2</td>
<td>C</td>
</tr>
<tr>
<td>2.3</td>
<td>C++</td>
</tr>
<tr>
<td>2.4</td>
<td>Fortran</td>
</tr>
<tr>
<td>2.5</td>
<td>Java</td>
</tr>
<tr>
<td>2.6</td>
<td>Maple</td>
</tr>
</tbody>
</table>

These routines

<table>
<thead>
<tr>
<th>Pair</th>
<th>Routines</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, 2)</td>
<td><code>euler2a, euler2b, sseuler</code></td>
</tr>
<tr>
<td>(2, 4)</td>
<td><code>rk4a, rk4b, ssrk4</code></td>
</tr>
<tr>
<td>(4, 5)</td>
<td><code>rkf45a, rkf45b, ssrkf45</code></td>
</tr>
</tbody>
</table>

...
3

Solution of a 2 x 2 ODE System

We now consider the programming of the 2x2 ODE problem of Equations 1.6, 1.16, and 1.17 using the general-purpose ODE library integrators from Chapter 2. Since these integrators were listed and discussed in some detail in Chapter 2, they are not discussed here, except as they are called in a main program for the solution of the 2x2 ODE problem. The order in which the six languages are considered is the same as in Chapter 2.

3.1 Programming in MATLAB

A main program for the solution of the 2x2 ODE problem is listed below:

```matlab
%% Main program ode2x2 computes the numerical solution to the 2 x 2 ODE system by six integrators
%% Step through six integrators
for int=1:6
    %% Integration parameters
    [neqn,nout,nsteps,t0,tf,abserr,relerr]=intpar;
    %% Initial condition vector
    [u0]=inital(neqn,t0);
    %% Output interval
    tp=tf-t0;
```
% Compute solution at nout output points
  for j=1:nout
    % Print current solution
    [out]=fprint(int,neqn,t0,u0);
    % Fixed step modified Euler integrator
    if int == 1
      [u]=euler2a(neqn,t0,tf,u0,nsteps);
    end
    % Variable step modified Euler integrator
    if int == 2
      [u]=euler2b(neqn,t0,tf,u0,nsteps,abserr,relerr);
    end
    % Fixed step classical fourth order RK integrator
    if int == 3
      [u]=rkc4a(neqn,t0,tf,u0,nsteps);
    end
    % Variable step classical fourth order RK integrator
    if int == 4
      [u]=rkc4b(neqn,t0,tf,u0,nsteps,abserr,relerr);
    end
    % Fixed step RK Fehlberg (RKF45) integrator
    if int == 5
      [u]=rkf45a(neqn,t0,tf,u0,nsteps);
    end
    % Variable step RK Fehlberg (RKF45) integrator
    if int == 6
      [u]=rkf45b(neqn,t0,tf,u0,nsteps,abserr,relerr);
    end
    % Advance solution
    t0=tf;
    tf=tf+tp;
    u0=u;
  % Next output
end
Program 3.1.1
MATLAB main program for the numerical integration of Equations 1.6, 1.16, with analytical solution Equation 1.17

The only essential difference between Program 3.1.1 and 2.1.1 is the use of a loop in the latter to cycle through all six integrators:

```matlab
% Step through six integrators
for int=1:6
    int, which was set for a particular integrator by a call to intpar in Program 2.1.1, is now set by this for loop.
    Routines intpar, initial, deriv, and fprintf are listed below:
```

```matlab
function [neqn,nout,nsteps,t0,tf,abserr,relerr]=intpar
    % Function intpar sets the parameters to control the integration of the 2 x 2 ODE system
    % Number of first order ODEs
    neqn=2;
    % Number of output points
    nout=6;
    % Maximum number of steps in the interval t0 to tf
    nsteps=100;
    % Initial, final values of independent variable
    t0=0.0;
    tf=1.0;
    % Error tolerances
    abserr=1.0e-05;
    relerr=1.0e-05;
    %
    % End of intpar
```
function [u0]=inital(neqn,t)
if
% Function inital sets the initial condition vector
% for the 2 x 2 ODE problem
% 
% u0(1)=0;
% u0(2)=2;
% 
% End of inital

function [ut]=derv(neqn,t,u)
if
% Function derv computes the derivative vector
% of the 2 x 2 ODE problem
% 
% Problem parameters
% a=5.5;
% b=4.5;
% 
% Derivative vector
% ut(1)=-a*u(1)+b*u(2);
% ut(2)= b*u(1)-a*u(2);
% 
% End of derv

function [out]=fprint(ncase,neqn,t,u)
if
% Function fprint displays the numerical and
% exact solutions to the 2 x 2 ODE problem
% 
% Return current value of independent variable
% (MATLAB requires at least one return argument)
% out=t;
% 
% Problem parameters
% a=5.5;
% b=4.5;
% 
% Print a heading for the solution at t = 0
% if(t<=0.0)
% Label for ODE integrator
% Fixed step modified Euler
if(ncase==1)
    fprintf(' euler2a integrator
');
% Variable step modified Euler
elseif(ncase==2)
    fprintf(' euler2b integrator
');
% Fixed step classical fourth order RK
elseif(ncase==3)
    fprintf(' rkc4a integrator
');
% Variable step classical fourth order RK
elseif(ncase==4)
    fprintf(' rkc4b integrator
');
% Fixed step RK Fehlberg 45
elseif(ncase==5)
    fprintf(' rkf45a integrator
');
% Variable step RK Fehlberg 45
elseif(ncase==6)
    fprintf(' rkf45b integrator
');
end
% Heading
fprintf(' t u1 u2 u1-ue1 u2-ue2
');
% End of t = 0 heading
end
% Numerical and analytical solution output

% Exact solution eigenvalues
e1=-(a-b);
e2=-(a+b);
% Analytical solution
ue(1)=exp(e1*t)-exp(e2*t);
ue(2)=exp(e1*t)+exp(e2*t);
% Difference between exact and numerical solutions
diff=u-ue;
We can note the following points about these routines:

- The initial conditions of Equations 1.6 are set in `inital` as \( y_1(0) = y_{10} = 0, \)
  \( y_2(0) = y_{20} = 2: \)
  ```
  u0(1)=0;
  u0(2)=2;
  ```

- The RHS of Equations 1.6 are programmed in `derv` as
  ```
  % Problem parameters
  a=5.5;
  b=4.5;
  % Derivative vector
  ut(1)=-a*u(1)+b*u(2);
  ut(2)= b*u(1)-a*u(2);
  ```

- The analytical solution, Equation 1.17, is programmed in `fprint` as
  ```
  % Exact solution eigenvalues
  e1=-(a-b);
  e2=-(a+b);
  % Analytical solution
  ue(1)=exp(e1*t)-exp(e2*t);
  ue(2)=exp(e1*t)+exp(e2*t);
  ```

- The numerical and analytical solutions, and their difference, are then displayed in `fprint`:
  ```
  % Difference between exact and numerical solutions
  diff=u-ue;
  ```
Display the numerical and exact solutions, and their difference:

```matlab
fprintf('%10.2f %10.5f %10.5f %10.5f %10.5f 
', t, u, diff);
```

Since the library integration routines called by Program 3.1.1, `euler2a`, `euler2b`, `sseuler`, `rkc4a`, `rkc4b`, `ssrkc4`, `rkf45a`, `rkf45ba`, and `ssrkf45`, are considered in detail in Chapter 2, they will not be discussed here.

The output from Programs 3.1.1 and 3.1.2 is listed below:

**euler2a integrator**

<table>
<thead>
<tr>
<th>t</th>
<th>u1</th>
<th>u2</th>
<th>u1-ue1</th>
<th>u2-ue2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.00000</td>
<td>2.00000</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>1.00</td>
<td>0.36784</td>
<td>0.36793</td>
<td>0.00001</td>
<td>0.00001</td>
</tr>
<tr>
<td>2.00</td>
<td>0.13534</td>
<td>0.13534</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>3.00</td>
<td>0.04979</td>
<td>0.04979</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>4.00</td>
<td>0.01832</td>
<td>0.01832</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>5.00</td>
<td>0.00674</td>
<td>0.00674</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
</tbody>
</table>

**euler2b integrator**

<table>
<thead>
<tr>
<th>t</th>
<th>u1</th>
<th>u2</th>
<th>u1-ue1</th>
<th>u2-ue2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.00000</td>
<td>2.00000</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>1.00</td>
<td>0.36783</td>
<td>0.36792</td>
<td>0.00001</td>
<td>0.00001</td>
</tr>
<tr>
<td>2.00</td>
<td>0.13534</td>
<td>0.13534</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>3.00</td>
<td>0.04979</td>
<td>0.04979</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>4.00</td>
<td>0.01832</td>
<td>0.01832</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>5.00</td>
<td>0.00674</td>
<td>0.00674</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
</tbody>
</table>

**rkc4a integrator**

<table>
<thead>
<tr>
<th>t</th>
<th>u1</th>
<th>u2</th>
<th>u1-ue1</th>
<th>u2-ue2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.00000</td>
<td>2.00000</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>1.00</td>
<td>0.36783</td>
<td>0.36792</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>2.00</td>
<td>0.13534</td>
<td>0.13534</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>3.00</td>
<td>0.04979</td>
<td>0.04979</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>4.00</td>
<td>0.01832</td>
<td>0.01832</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>5.00</td>
<td>0.00674</td>
<td>0.00674</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
</tbody>
</table>
We conclude from this output that the error tolerances set in \textit{intpar} \((1.0 \times 10^{-5})\) are observed by all six integrators.

\section*{3.2 Programming in C}

Since main Program 2.2.1 and the associated header file Program 2.2.2 are unchanged in the 2x2 ODE problem, they are not listed here. \textit{intpar}, \textit{par}, \textit{inital}, \textit{derv}, and \textit{fprint} are listed below:
#include "ode2x2.h"

/* Type global variables */
int neqn, nout, nsteps;
double t0, tf, abserr, relerr;

/* Define file ID */
FILE *fid;

void intpar()

/* Function intpar sets the parameters to control the integration of the 2 x 2 ODE system */
{
    /* Number of ODEs */
    neqn=2;

    /* Number of output points */
    nout=6;

    /* Maximum number of steps in the interval t0 to tf */
    nsteps=100;

    /* Initial, final values of independent variable */
    t0=0.0;
    tf=1.0;

    /* Error tolerances */
    abserr=pow(10,-5);
    relerr=pow(10,-5);

    /* End of intpar */
}

void par(double a[])

/* Function par sets the parameters for the 2 x 2 ODE problem */
{
    a[1]=5.5;
    a[2]=4.5;
void inital(double u0[], double t0)

/* Function inital sets the initial condition vector for
the 2 x 2 ODE problem */
{
    /* Initial condition */
    u0[1] = 0.0;
    u0[2] = 2.0;

    /* End of inital */
}

void derv(double ut[], double t, double u[])

/* Function derv computes the derivative vector of the
2 x 2 ODE problem */
{
    /* Type variables */
    double a[3];

    /* Problem parameters */
    par(a);

    /* Derivative vector */

    /* End of derv */
}

void fprint(int ncase, double t, double u[])

/* Function fprint displays the numerical and exact
solutions to the 2 x 2 ODE problem */
{
    /* Type variables */
    double a[3], ue[3], diff[3], e1, e2;

/* Problem parameters */
par(a);

/* Print a heading for the solution at t = 0 */
if(t<=0.0)
{
    /* Label for ODE integrator */
    switch(ncase)
    {
        case 1: /* Fixed step modified Euler */
            fprintf(fid,"\n   euler2a integrator\n");
            break;
        case 2: /* Variable step modified Euler */
            fprintf(fid,"\n   euler2b integrator\n");
            break;
        case 3: /* Fixed step classical fourth order RK */
            fprintf(fid,"\n   rkc4a integrator\n");
            break;
        case 4: /* Variable step classical fourth order RK */
            fprintf(fid,"\n   rkc4b integrator\n");
            break;
        case 5: /* Fixed step RK Fehlberg 45 */
            fprintf(fid,"\n   rkf45a integrator\n");
            break;
        case 6: /* Variable step RK Fehlberg 45 */
            fprintf(fid,"\n   rkf45b integrator\n");
            break;
    }
    /* Heading */
    fprintf(fid,"\n   t   u1(num)   u1(ex)    diff1\n   u2(num)   u2(ex)    diff2\n"");
    fprintf(fid,"\n"");

    /* End of t = 0 heading */
}

/* Analytical solution eigenvalues */
e1=-(a[1]-a[2]);
e2=-(a[1]+a[2]);
/* Analytical solution vector */
ue[1]=exp(e1*t)-exp(e2*t);
ue[2]=exp(e1*t)+exp(e2*t);

/* Difference between exact and numerical solution vectors */
diff[1]=u[1]-ue[1];

/* Display the numerical and exact solutions, and their difference */
fprintf(fid,"%10.2f %10.5f %10.5f %13.4e \n",
t,u[1],ue[1],diff[1]);
fprintf(fid,"%10.5f %10.5f %13.4e \n\n",
u[2],ue[2],diff[2]);

/* End of fprintf */
}

Program 3.2.1
intpar, par, initial, derv, and fprintf for the solution of Equations 1.6 and 1.16

The only new feature of these routines is the addition of par, which sets the problem parameters \( a = 5.5, b = 4.5 \). These parameters are then used in derv and fprintf by a call to par:

/* Type variables */
double a[3];

/* Problem parameters */
par(a);

Of course, these parameters could be set directly in derv and fprintf as in Program 3.1.2. The use of par is just an alternative, which would be more attractive as the number of parameters becomes large (so that the code for the assignment statements is programmed once in par, then used in more than one place, such as in derv and fprintf, by calls to par).

The output from the preceding routines is as follows:

euler2a integrator
\[
\begin{array}{cccc}
  t & u1(num) & u1(ex) & diff1 \\
  u2(num) & u2(ex) & diff2 \\
  0.00 & 0.00000 & 0.00000 & 0.00000e+00 \\
  2.00000 & 2.00000 & 0.00000e+00 \\
\end{array}
\]
<table>
<thead>
<tr>
<th>t</th>
<th>u1(num)</th>
<th>u1(ex)</th>
<th>diff1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00</td>
<td>0.36784</td>
<td>0.36783</td>
<td>5.3545e-06</td>
</tr>
<tr>
<td></td>
<td>0.36793</td>
<td>0.36792</td>
<td>7.0006e-06</td>
</tr>
<tr>
<td>2.00</td>
<td>0.13534</td>
<td>0.13534</td>
<td>4.5451e-06</td>
</tr>
<tr>
<td></td>
<td>0.13534</td>
<td>0.13534</td>
<td>4.5453e-06</td>
</tr>
<tr>
<td>3.00</td>
<td>0.04979</td>
<td>0.04979</td>
<td>2.5082e-06</td>
</tr>
<tr>
<td></td>
<td>0.04979</td>
<td>0.04979</td>
<td>2.5082e-06</td>
</tr>
<tr>
<td>4.00</td>
<td>0.01832</td>
<td>0.01832</td>
<td>1.2303e-06</td>
</tr>
<tr>
<td></td>
<td>0.01832</td>
<td>0.01832</td>
<td>1.2303e-06</td>
</tr>
<tr>
<td>5.00</td>
<td>0.00674</td>
<td>0.00674</td>
<td>5.6575e-07</td>
</tr>
<tr>
<td></td>
<td>0.00674</td>
<td>0.00674</td>
<td>5.6575e-07</td>
</tr>
</tbody>
</table>

**euler2b integrator**

<table>
<thead>
<tr>
<th>t</th>
<th>u1(num)</th>
<th>u1(ex)</th>
<th>diff1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.0000e+00</td>
</tr>
<tr>
<td></td>
<td>2.00000</td>
<td>2.00000</td>
<td>0.0000e+00</td>
</tr>
<tr>
<td>1.00</td>
<td>0.36784</td>
<td>0.36783</td>
<td>5.4556e-06</td>
</tr>
<tr>
<td></td>
<td>0.36793</td>
<td>0.36792</td>
<td>7.1360e-06</td>
</tr>
<tr>
<td>2.00</td>
<td>0.13534</td>
<td>0.13534</td>
<td>4.6322e-06</td>
</tr>
<tr>
<td></td>
<td>0.13534</td>
<td>0.13534</td>
<td>4.6323e-06</td>
</tr>
<tr>
<td>3.00</td>
<td>0.04979</td>
<td>0.04979</td>
<td>2.6575e-06</td>
</tr>
<tr>
<td></td>
<td>0.04979</td>
<td>0.04979</td>
<td>2.6575e-06</td>
</tr>
<tr>
<td>4.00</td>
<td>0.01832</td>
<td>0.01832</td>
<td>1.9472e-06</td>
</tr>
<tr>
<td></td>
<td>0.01832</td>
<td>0.01832</td>
<td>1.9472e-06</td>
</tr>
<tr>
<td>5.00</td>
<td>0.00674</td>
<td>0.00674</td>
<td>1.8392e-06</td>
</tr>
<tr>
<td></td>
<td>0.00674</td>
<td>0.00674</td>
<td>1.8392e-06</td>
</tr>
</tbody>
</table>

**rkc4a integrator**

<table>
<thead>
<tr>
<th>t</th>
<th>u1(num)</th>
<th>u1(ex)</th>
<th>diff1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.0000e+00</td>
</tr>
<tr>
<td></td>
<td>2.00000</td>
<td>2.00000</td>
<td>0.0000e+00</td>
</tr>
<tr>
<td>t</td>
<td>u1(num)</td>
<td>u1(ex)</td>
<td>diff1</td>
</tr>
<tr>
<td>-----</td>
<td>---------</td>
<td>----------</td>
<td>--------</td>
</tr>
<tr>
<td>1.00</td>
<td>0.36783</td>
<td>0.36783</td>
<td>-3.8034e-10</td>
</tr>
<tr>
<td></td>
<td>0.36792</td>
<td>0.36792</td>
<td>4.4217e-10</td>
</tr>
<tr>
<td>2.00</td>
<td>0.13534</td>
<td>0.13534</td>
<td>2.2707e-11</td>
</tr>
<tr>
<td></td>
<td>0.13534</td>
<td>0.13534</td>
<td>2.2782e-11</td>
</tr>
<tr>
<td>3.00</td>
<td>0.04979</td>
<td>0.04979</td>
<td>1.2551e-11</td>
</tr>
<tr>
<td></td>
<td>0.04979</td>
<td>0.04979</td>
<td>1.2551e-11</td>
</tr>
<tr>
<td>4.00</td>
<td>0.01832</td>
<td>0.01832</td>
<td>6.1563e-12</td>
</tr>
<tr>
<td></td>
<td>0.01832</td>
<td>0.01832</td>
<td>6.1563e-12</td>
</tr>
<tr>
<td>5.00</td>
<td>0.00674</td>
<td>0.00674</td>
<td>2.8310e-12</td>
</tr>
<tr>
<td></td>
<td>0.00674</td>
<td>0.00674</td>
<td>2.8310e-12</td>
</tr>
</tbody>
</table>

rkc4b integrator

<table>
<thead>
<tr>
<th>t</th>
<th>u1(num)</th>
<th>u1(ex)</th>
<th>diff1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.0000e+00</td>
</tr>
<tr>
<td>2.00000</td>
<td>2.00000</td>
<td>0.0000e+00</td>
<td></td>
</tr>
<tr>
<td>1.00</td>
<td>0.36783</td>
<td>0.36783</td>
<td>-1.8012e-08</td>
</tr>
<tr>
<td></td>
<td>0.36792</td>
<td>0.36792</td>
<td>2.0370e-08</td>
</tr>
<tr>
<td>2.00</td>
<td>0.13534</td>
<td>0.13534</td>
<td>1.5347e-09</td>
</tr>
<tr>
<td></td>
<td>0.13534</td>
<td>0.13534</td>
<td>1.5407e-09</td>
</tr>
<tr>
<td>3.00</td>
<td>0.04979</td>
<td>0.04979</td>
<td>7.2352e-09</td>
</tr>
<tr>
<td></td>
<td>0.04979</td>
<td>0.04979</td>
<td>7.2352e-09</td>
</tr>
<tr>
<td>4.00</td>
<td>0.01832</td>
<td>0.01832</td>
<td>5.1152e-09</td>
</tr>
<tr>
<td></td>
<td>0.01832</td>
<td>0.01832</td>
<td>5.1152e-09</td>
</tr>
<tr>
<td>5.00</td>
<td>0.00674</td>
<td>0.00674</td>
<td>1.7098e-08</td>
</tr>
<tr>
<td></td>
<td>0.00674</td>
<td>0.00674</td>
<td>1.7098e-08</td>
</tr>
</tbody>
</table>

rkf45a integrator

<table>
<thead>
<tr>
<th>t</th>
<th>u1(num)</th>
<th>u1(ex)</th>
<th>diff1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.0000e+00</td>
</tr>
<tr>
<td></td>
<td>2.00000</td>
<td>2.00000</td>
<td>0.0000e+00</td>
</tr>
<tr>
<td>t</td>
<td>u1(num)</td>
<td>u1(ex)</td>
<td>diff1</td>
</tr>
<tr>
<td>----</td>
<td>---------</td>
<td>--------</td>
<td>---------</td>
</tr>
<tr>
<td>1.00</td>
<td>0.36783</td>
<td>0.36783</td>
<td>4.4252e-12</td>
</tr>
<tr>
<td></td>
<td>0.36792</td>
<td>0.36792</td>
<td>-4.4910e-12</td>
</tr>
<tr>
<td>2.00</td>
<td>0.13534</td>
<td>0.13534</td>
<td>-2.4120e-14</td>
</tr>
<tr>
<td></td>
<td>0.13534</td>
<td>0.13534</td>
<td>-2.4869e-14</td>
</tr>
<tr>
<td>3.00</td>
<td>0.04979</td>
<td>0.04979</td>
<td>-1.3572e-14</td>
</tr>
<tr>
<td></td>
<td>0.04979</td>
<td>0.04979</td>
<td>-1.3593e-14</td>
</tr>
<tr>
<td>4.00</td>
<td>0.01832</td>
<td>0.01832</td>
<td>-6.6648e-15</td>
</tr>
<tr>
<td></td>
<td>0.01832</td>
<td>0.01832</td>
<td>-6.6648e-15</td>
</tr>
<tr>
<td>5.00</td>
<td>0.00674</td>
<td>0.00674</td>
<td>-3.0739e-15</td>
</tr>
<tr>
<td></td>
<td>0.00674</td>
<td>0.00674</td>
<td>-3.0739e-15</td>
</tr>
</tbody>
</table>

rKF45b integrator

<table>
<thead>
<tr>
<th>t</th>
<th>u1(num)</th>
<th>u1(ex)</th>
<th>diff1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.0000e+00</td>
</tr>
<tr>
<td>2.00000</td>
<td>2.00000</td>
<td>0.0000e+00</td>
<td></td>
</tr>
<tr>
<td>1.00</td>
<td>0.36783</td>
<td>0.36783</td>
<td>8.6412e-07</td>
</tr>
<tr>
<td></td>
<td>0.36792</td>
<td>0.36792</td>
<td>-8.7008e-07</td>
</tr>
<tr>
<td>2.00</td>
<td>0.13534</td>
<td>0.13534</td>
<td>-1.4522e-07</td>
</tr>
<tr>
<td></td>
<td>0.13534</td>
<td>0.13534</td>
<td>-1.4887e-07</td>
</tr>
<tr>
<td>3.00</td>
<td>0.04978</td>
<td>0.04979</td>
<td>-2.1496e-06</td>
</tr>
<tr>
<td></td>
<td>0.04978</td>
<td>0.04979</td>
<td>-2.1386e-06</td>
</tr>
<tr>
<td>4.00</td>
<td>0.01831</td>
<td>0.01832</td>
<td>-1.6852e-06</td>
</tr>
<tr>
<td></td>
<td>0.01831</td>
<td>0.01832</td>
<td>-1.4300e-06</td>
</tr>
<tr>
<td>5.00</td>
<td>0.00673</td>
<td>0.00674</td>
<td>-3.8224e-06</td>
</tr>
<tr>
<td></td>
<td>0.00674</td>
<td>0.00674</td>
<td>2.1107e-06</td>
</tr>
</tbody>
</table>

Generally, the accuracy of the numerical solution meets or exceeds the tolerances set in intpar.
3.3 Programming in C++

Again, since main Program 2.3.1 and the associated header file Program 2.3.2 are unchanged in the 2x2 ODE problem, they are not listed here. intpar, par, inital, derv, and fprint are listed below:

```cpp
#include *DEF.h*
#include <iomanip.h>

/* Define file ID */
FILE *fid;

void DEF::intpar()

/* Function intpar sets the parameters to control the integration of the 2 x 2 ODE system */
{
    /* Number of ODEs */
    neqn=2;

    /* Number of output points */
    nout=6;

    /* Maximum number of steps in the interval t0 to tf */
    nsteps=100;

    /* Initial, final values of independent variable */
    t0=0.0;
    tf=1.0;

    /* Error tolerances */
    abserr=pow(10.0,-5.0);
    relerr=pow(10.0,-5.0);

    /* End of intpar */
}

void DEF::inital()

/* Function inital sets the initial condition vector for the 2 x 2 ODE problem */
```
{  /* Initial condition */  u0[1]=0.0;  u0[2]=2.0;  /* End of initial */ }

void DEF::derv(double ut[], double t, double u[])
/* Function derv computes the derivative vector of the 2 x 2 ODE problem */
{
  /* Type variables */
  double a, b;

  /* Problem parameters */
  a=5.5;
  b=4.5;

  /* Derivative vector */
  ut[2]= b*u[1]-a*u[2];

  /* End of derv */
}

void DEF::fprint(int ncase, int neqn, double t, double u[])
/* Function fprint displays the numerical and exact solutions to the 2 x 2 ODE problem; this function has two override-defined functions */
{
  /* Type variables */
  double ue[3], diff[3];
  double a, b, e1, e2;

  /* Problem parameters */
  a=5.5;
  b=4.5;
/* Print a heading for the solution at t = 0 */
if (t <= 0.0)
{
  /* Label for ODE integrator */
  switch (ncase)
  {
    /*Fixed step modified Euler */
    case 1:
      fprintf(fid, "\n\n euler2a integrator\n\n");
      break;

    /* Variable step modified Euler */
    case 2:
      fprintf(fid, "\n\n euler2b integrator\n\n");
      break;

    /* Fixed step classical fourth order RK */
    case 3:
      fprintf(fid, "\n\n rkc4a integrator\n\n");
      break;

    /* Variable step classical fourth order RK */
    case 4:
      fprintf(fid, "\n\n rkc4b integrator\n\n");
      break;

    /* Fixed step RK Fehlberg 45 */
    case 5:
      fprintf(fid, "\n\n rkf45a integrator\n\n");
      break;

    /* Variable step RK Fehlberg 45 */
    case 6:
      fprintf(fid, "\n\n rkf45b integrator\n\n");
      break;

  }

  /* Heading */
  fprintf(fid, "\n\n t \ u1(num) \ u1(ex) \ diff1\n");
  fprintf(fid, "\n\n \ u2(num) \ u2(ex) \ diff2\n");

  /* End of t = 0 heading */
}

Copyright 2004 by Chapman & Hall/CRC
/* Analytical solution eigenvalues */
e1=-(a-b);
e2=-(a+b);

/* Analytical solution vector */
ue[1]=exp(e1*t)-exp(e2*t);
ue[2]=exp(e1*t)+exp(e2*t);

/* Difference between exact and numerical solutions */
diff[1]=u[1]-ue[1];

/* Display the numerical and exact solutions, and their difference */
fprintf(fid,"%10.2f %10.5f %10.5f %13.4e
",t,u[1],ue[1],
diff[1]);
fprintf(fid,"%10.2f %10.5f %10.5f %13.4e
",t,u[2],ue[2],
diff[2]);

/* End of fprint */
}

doctor void DEF::fprint(ofstream &fout, int ncase, int neqn,
double t, double u[])

/* Function fprint displays the numerical and exact solutions to the 2 x 2 ODE problem; this function has
two override-defined functions */
{
    /* Type variables */
    double ue[3], diff[3];
double a, b, e1, e2;

    /* Problem parameters */
a=5.5;
b=4.5;

    /* Set printing format */
    fout<<setiosflags(ios::showpoint|ios::fixed)
        <<setprecision(7);

    /* Print a heading for the solution at t = 0 */
    if(t<=0.0)
    {
/* Label for ODE integrator */
switch(ncase)
{
    /* Fixed step modified Euler */
    case 1:
        fout<<"\n\neuler2a integrator\n";
        break;

    /* Variable step modified Euler */
    case 2:
        fout<<"\n\neuler2b integrator\n";
        break;

    /* Fixed step classical fourth order RK */
    case 3:
        fout<<"\n\nrkc4a integrator\n";
        break;

    /* Variable step classical fourth order RK */
    case 4:
        fout<<"\n\nrkc4b integrator\n";
        break;

    /* Fixed step RK Fehlberg 45 */
    case 5:
        fout<<"\n\nrkf45a integrator\n";
        break;

    /* Variable step RK Fehlberg 45 */
    case 6:
        fout<<"\n\nrkf45b integrator\n";
        break;
}

/* Heading */
fout<<endl;
fout<<" t"<<setw(18)<<"u1(num)"<<setw(11)
    <<"u1(ex)"<<setw(11)<<"diff1"<<"\n";
fout    <<setw(20)<<"u2(num)"<<setw(11)
    <<"u2(ex)"<<setw(11)<<"diff2"<<"\n";

/* End of t = 0 heading */
}
/ * Analytical solution eigenvalues */
 e1 = -(a - b);
 e2 = -(a + b);

 /* Analytical solution vector */
 ue[1] = exp(e1*t) - exp(e2*t);
 ue[2] = exp(e1*t) + exp(e2*t);

 /* Difference between exact and numerical solutions */
 fout << endl;

 /* Display the numerical and exact solutions, and their difference */

 /* End of fprint */
}

Program 3.3.1
intpar, initial, derv, and fprint for the solution of Equations 1.6 and 1.16

The output from the preceding routines is as follows:

euler2a integrator

<table>
<thead>
<tr>
<th>t</th>
<th>u1(num)</th>
<th>u1(ex)</th>
<th>diff1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
</tr>
<tr>
<td>2.000000</td>
<td>2.000000</td>
<td>2.000000</td>
<td>0.000000</td>
</tr>
<tr>
<td>1.000000</td>
<td>0.3678394</td>
<td>0.3678340</td>
<td>0.0000054</td>
</tr>
<tr>
<td>0.3679318</td>
<td>0.3679248</td>
<td>0.0000070</td>
<td></td>
</tr>
<tr>
<td>2.000000</td>
<td>0.1353398</td>
<td>0.1353353</td>
<td>0.0000045</td>
</tr>
<tr>
<td>0.1353398</td>
<td>0.1353353</td>
<td>0.0000045</td>
<td></td>
</tr>
<tr>
<td>3.000000</td>
<td>0.0497896</td>
<td>0.0497871</td>
<td>0.0000025</td>
</tr>
<tr>
<td>0.0497896</td>
<td>0.0497871</td>
<td>0.0000025</td>
<td></td>
</tr>
<tr>
<td>t</td>
<td>u1(num)</td>
<td>u1(ex)</td>
<td>diff1</td>
</tr>
<tr>
<td>-----</td>
<td>---------</td>
<td>--------</td>
<td>-------</td>
</tr>
<tr>
<td>0.000000</td>
<td>0.0000000</td>
<td>0.0000000</td>
<td>0.0000000</td>
</tr>
<tr>
<td>1.000000</td>
<td>0.3678395</td>
<td>0.3678340</td>
<td>0.0000055</td>
</tr>
<tr>
<td>2.000000</td>
<td>0.1353399</td>
<td>0.1353353</td>
<td>0.0000046</td>
</tr>
<tr>
<td>3.000000</td>
<td>0.0497897</td>
<td>0.0497871</td>
<td>0.0000027</td>
</tr>
<tr>
<td>4.000000</td>
<td>0.0183169</td>
<td>0.0183156</td>
<td>0.0000012</td>
</tr>
<tr>
<td>5.000000</td>
<td>0.0067385</td>
<td>0.0067379</td>
<td>0.0000006</td>
</tr>
</tbody>
</table>

*euler2b integrator*

<table>
<thead>
<tr>
<th>t</th>
<th>u1(num)</th>
<th>u1(ex)</th>
<th>diff1</th>
<th>u2(num)</th>
<th>u2(ex)</th>
<th>diff2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.000000</td>
<td>0.0000000</td>
<td>0.0000000</td>
<td>0.0000000</td>
<td>2.0000000</td>
<td>2.0000000</td>
<td>0.0000000</td>
</tr>
<tr>
<td>1.000000</td>
<td>0.3678395</td>
<td>0.3678340</td>
<td>0.0000055</td>
<td>0.3679248</td>
<td>0.3679248</td>
<td>0.0000071</td>
</tr>
<tr>
<td>2.000000</td>
<td>0.1353399</td>
<td>0.1353353</td>
<td>0.0000046</td>
<td>0.1353399</td>
<td>0.1353353</td>
<td>0.0000046</td>
</tr>
<tr>
<td>3.000000</td>
<td>0.0497897</td>
<td>0.0497871</td>
<td>0.0000027</td>
<td>0.0497897</td>
<td>0.0497871</td>
<td>0.0000027</td>
</tr>
<tr>
<td>4.000000</td>
<td>0.0183169</td>
<td>0.0183156</td>
<td>0.0000012</td>
<td>0.0183169</td>
<td>0.0183156</td>
<td>0.0000012</td>
</tr>
<tr>
<td>5.000000</td>
<td>0.0067385</td>
<td>0.0067379</td>
<td>0.0000006</td>
<td>0.0067385</td>
<td>0.0067379</td>
<td>0.0000006</td>
</tr>
</tbody>
</table>

*rkc4a integrator*
rkc4b integrator

\[
\begin{array}{cccc}
3.000000 & 0.0497871 & 0.0497871 & 0.0000000 \\
0.0497871 & 0.0497871 & 0.0000000 \\
4.000000 & 0.0183156 & 0.0183156 & 0.0000000 \\
0.0183156 & 0.0183156 & 0.0000000 \\
5.000000 & 0.0067379 & 0.0067379 & 0.0000000 \\
0.0067379 & 0.0067379 & 0.0000000 \\
\end{array}
\]

rkc4b integrator

\[
\begin{array}{cccc}
t & u_1(\text{num}) & u_1(\text{ex}) & \text{diff1} \\
u_2(\text{num}) & u_2(\text{ex}) & \text{diff2} \\
0.000000 & 0.0000000 & 0.0000000 & 0.0000000 \\
2.0000000 & 2.0000000 & 0.0000000 \\
1.000000 & 0.3678340 & 0.3678340 & 0.0000000 \\
0.3679249 & 0.3679248 & 0.0000000 \\
2.000000 & 0.1353353 & 0.1353353 & 0.0000000 \\
0.1353353 & 0.1353353 & 0.0000000 \\
3.000000 & 0.0497871 & 0.0497871 & 0.0000000 \\
0.0497871 & 0.0497871 & 0.0000000 \\
4.000000 & 0.0183156 & 0.0183156 & 0.0000000 \\
0.0183156 & 0.0183156 & 0.0000000 \\
5.000000 & 0.0067380 & 0.0067379 & 0.0000000 \\
0.0067380 & 0.0067379 & 0.0000000 \\
\end{array}
\]

rkc4b integrator

\[
\begin{array}{cccc}
t & u_1(\text{num}) & u_1(\text{ex}) & \text{diff1} \\
u_2(\text{num}) & u_2(\text{ex}) & \text{diff2} \\
0.000000 & 0.0000000 & 0.0000000 & 0.0000000 \\
2.0000000 & 2.0000000 & 0.0000000 \\
1.000000 & 0.3678340 & 0.3678340 & 0.0000000 \\
0.3679248 & 0.3679248 & 0.0000000 \\
2.000000 & 0.1353353 & 0.1353353 & 0.0000000 \\
0.1353353 & 0.1353353 & 0.0000000 \\
3.000000 & 0.0497871 & 0.0497871 & 0.0000000 \\
0.0497871 & 0.0497871 & 0.0000000 \\
4.000000 & 0.0183156 & 0.0183156 & 0.0000000 \\
0.0183156 & 0.0183156 & 0.0000000 \\
5.000000 & 0.0067380 & 0.0067379 & 0.0000000 \\
0.0067380 & 0.0067379 & 0.0000000 \\
\end{array}
\]

rkc4b integrator

\[
\begin{array}{cccc}
t & u_1(\text{num}) & u_1(\text{ex}) & \text{diff1} \\
u_2(\text{num}) & u_2(\text{ex}) & \text{diff2} \\
0.000000 & 0.0000000 & 0.0000000 & 0.0000000 \\
2.0000000 & 2.0000000 & 0.0000000 \\
1.000000 & 0.3678340 & 0.3678340 & 0.0000000 \\
0.3679248 & 0.3679248 & 0.0000000 \\
2.000000 & 0.1353353 & 0.1353353 & 0.0000000 \\
0.1353353 & 0.1353353 & 0.0000000 \\
3.000000 & 0.0497871 & 0.0497871 & 0.0000000 \\
0.0497871 & 0.0497871 & 0.0000000 \\
4.000000 & 0.0183156 & 0.0183156 & 0.0000000 \\
0.0183156 & 0.0183156 & 0.0000000 \\
5.000000 & 0.0067380 & 0.0067379 & 0.0000000 \\
0.0067380 & 0.0067379 & 0.0000000 \\
\end{array}
\]

Copyright 2004 by Chapman & Hall/CRC
<table>
<thead>
<tr>
<th>t</th>
<th>u1(num)</th>
<th>u1(ex)</th>
<th>diff1</th>
<th>u2(num)</th>
<th>u2(ex)</th>
<th>diff2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0000000</td>
<td>0.0000000</td>
<td>0.0000000</td>
<td>0.0000000</td>
<td>2.0000000</td>
<td>2.0000000</td>
<td>0.0000000</td>
</tr>
<tr>
<td>1.0000000</td>
<td>0.3678340</td>
<td>0.3678340</td>
<td>0.0000000</td>
<td>0.3679248</td>
<td>0.3679248</td>
<td>0.0000000</td>
</tr>
<tr>
<td>2.0000000</td>
<td>0.1353353</td>
<td>0.1353353</td>
<td>0.0000000</td>
<td>0.1353353</td>
<td>0.1353353</td>
<td>0.0000000</td>
</tr>
<tr>
<td>3.0000000</td>
<td>0.0497871</td>
<td>0.0497871</td>
<td>0.0000000</td>
<td>0.0497871</td>
<td>0.0497871</td>
<td>0.0000000</td>
</tr>
<tr>
<td>4.0000000</td>
<td>0.0183156</td>
<td>0.0183156</td>
<td>0.0000000</td>
<td>0.0183156</td>
<td>0.0183156</td>
<td>0.0000000</td>
</tr>
<tr>
<td>5.0000000</td>
<td>0.0067379</td>
<td>0.0067379</td>
<td>0.0000000</td>
<td>0.0067379</td>
<td>0.0067379</td>
<td>0.0000000</td>
</tr>
</tbody>
</table>

Generally, the accuracy of the numerical solution meets or exceeds the tolerances set in *intpar*.

### 3.4 Programming in Fortran

Again, since main Program 2.4.1 is unchanged in the 2x2 ODE problem, it is not listed here. *intpar, par, initial, derv, and fprint* are listed below:
subroutine intpar(neqn,nout,nsteps,t0,tf,abserr,relerr)
   C
   C Subroutine intpar sets the parameters to control the
   C integration of the 2 x 2 ODE system
   C
   C Double precision coding is used
       implicit double precision(a-h,o-z)
   C
   C Number of ODEs
       neqn=2
   C
   C Number of output points
       nout=6
   C
   C Maximum number of steps in the interval t0 to tf
       nsteps=100
   C
   C Initial, final values of the independent variable
       t0=0.0d0
       tf=0.2d0
   C
   C Error tolerances
       abserr=1.0d-05
       relerr=1.0d-05
   return
   C
   C End of intpar
   end

subroutine inital(neqn,t,u0)
   C
   C Subroutine inital sets the initial condition vector
   C for the 2 x 2 ODE problem
   C
   C Double precision coding is used
       implicit double precision(a-h,o-z)
   C
   C Size the arrays
       dimension u0(neqn)
   C
   C Initial condition
       u0(1)=0.0d0
       u0(2)=2.0d0
   return
C
C End of initial
end

subroutine par(a,b)
C
C Subroutine par sets the parameters for the 2 x 2 ODE problem
C
C Double precision coding is used
   implicit double precision(a-h,o-z)
C
C Problem parameters
   a=5.5d0
   b=4.5d0
   return
C
C End of par
end

subroutine deriv(neqn,t,u,ut)
C
C Subroutine deriv computes the derivative vector of the 2 x 2 ODE problem
C
C Double precision coding is used
   implicit double precision(a-h,o-z)
C
C Size the arrays
   dimension u(neqn), ut(neqn)
C
C Problem parameters
   call par(a,b)
C
C Derivative vector
   ut(1)=-a*u(1)+b*u(2)
   ut(2)= b*u(1)-a*u(2)
   return
C
C End of deriv
end
subroutine fprint(no, ncase, neqn, t, u)
C
C Subroutine fprint displays the numerical and
C analytical solutions to the 2 x 2 ODE problem
C Double precision coding is used
    implicit double precision(a-h,o-z)
C
C Size the arrays
    dimension u(neqn)
C
C Problem parameters
    call par(a,b)
C
C Print a heading for the solution at t = 0
    if(t.le.0.0d0)then
C
C    Label for ODE integrator
C
C    Fixed step modified Euler
    if(ncase.eq.1)then
        write(no,11)
    11 format(/,6x,'euler2a integrator')
C
C    Variable step modified Euler
    else if(ncase.eq.2)then
        write(no,12)
    12 format(/,6x,'euler2b integrator')
C
C    Fixed step classical fourth order RK
    else if(ncase.eq.3)then
        write(no,13)
    13 format(/,6x,'rkc4a integrator')
C
C    Variable step classical fourth order RK
    else if(ncase.eq.4)then
        write(no,14)
    14 format(/,6x,'rkc4b integrator')
C
C    Fixed step RK Fehlberg 45
    else if(ncase.eq.5)then
        write(no,15)
    15 format(/,6x,'rkf45a integrator')
C
C Variable step RK Fehlberg 45
else if(ncase.eq.6)then
  write(no,16)
16 format(/,6x,'rkf45b integrator')
end if
C
C Heading
  write(no,2)
2 format(/,9x,'t',3x,'u1(num)',4x,'u1(ex)',8x,'diff1',/,10x,3x,'u2(num)',4x,'u2(ex)',8x,'diff2',/)
C
C End of t = 0 heading
end if
C
C Analytical solution
  u1exact=dexp(-(a-b)*t)-dexp(-(a+b)*t)
  u2exact=dexp(-(a-b)*t)+dexp(-(a+b)*t)
C
C Difference between exact and numerical solution vectors
  diff1=u(1)-u1exact
  diff2=u(2)-u2exact
C
C Display the numerical and exact solutions,
C and their difference
  write(no,3)t,u(1),u1exact,diff1,u(2),u2exact,diff2
3 format(f10.2,2f10.5,e13.4,/,10x,2f10.5,e13.4,/) return
C
C End of fprint
end

Program 3.4.1
intpar, initial, par, derv, and fprint for the solution of Equations 1.6 and 1.16

The output from the preceding routines is as follows:

  euler2a integrator

<table>
<thead>
<tr>
<th>t</th>
<th>u1(num)</th>
<th>u1(ex)</th>
<th>diff1</th>
<th>diff2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.00000</td>
<td>0.0000</td>
<td>0.0000E+00</td>
<td>0.0000E+00</td>
</tr>
<tr>
<td>2.0000</td>
<td>2.0000</td>
<td>0.0000E+00</td>
<td>0.0000E+00</td>
<td></td>
</tr>
<tr>
<td>1.00</td>
<td>0.36784</td>
<td>0.36783</td>
<td>0.5354E-05</td>
<td>0.7001E-05</td>
</tr>
<tr>
<td>0.36793</td>
<td>0.36792</td>
<td>0.7001E-05</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Copyright 2004 by Chapman & Hall/CRC
2.00  0.13534  0.13534  0.4545E-05
   0.13534  0.13534  0.4545E-05
3.00  0.04979  0.04979  0.2508E-05
   0.04979  0.04979  0.2508E-05
4.00  0.01832  0.01832  0.1230E-05
   0.01832  0.01832  0.1230E-05
5.00  0.00674  0.00674  0.5657E-06
   0.00674  0.00674  0.5657E-06

euler2b integrator

<table>
<thead>
<tr>
<th>t</th>
<th>u1(num)</th>
<th>u1(ex)</th>
<th>diff1</th>
<th>u2(num)</th>
<th>u2(ex)</th>
<th>diff2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.0000E+00</td>
<td>2.00000</td>
<td>2.00000</td>
<td>0.0000E+00</td>
</tr>
<tr>
<td>1.00</td>
<td>0.36784</td>
<td>0.36783</td>
<td>0.5456E-05</td>
<td>0.36793</td>
<td>0.36792</td>
<td>0.7136E-05</td>
</tr>
<tr>
<td>2.00</td>
<td>0.13534</td>
<td>0.13534</td>
<td>0.4632E-05</td>
<td>0.13534</td>
<td>0.13534</td>
<td>0.4632E-05</td>
</tr>
<tr>
<td>3.00</td>
<td>0.04979</td>
<td>0.04979</td>
<td>0.2658E-05</td>
<td>0.04979</td>
<td>0.04979</td>
<td>0.2658E-05</td>
</tr>
<tr>
<td>4.00</td>
<td>0.01832</td>
<td>0.01832</td>
<td>0.1947E-05</td>
<td>0.01832</td>
<td>0.01832</td>
<td>0.1947E-05</td>
</tr>
<tr>
<td>5.00</td>
<td>0.00674</td>
<td>0.00674</td>
<td>0.1839E-05</td>
<td>0.00674</td>
<td>0.00674</td>
<td>0.1839E-05</td>
</tr>
</tbody>
</table>

rkc4a integrator

<table>
<thead>
<tr>
<th>t</th>
<th>u1(num)</th>
<th>u1(ex)</th>
<th>diff1</th>
<th>u2(num)</th>
<th>u2(ex)</th>
<th>diff2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.0000E+00</td>
<td>2.00000</td>
<td>2.00000</td>
<td>0.0000E+00</td>
</tr>
<tr>
<td>1.00</td>
<td>0.36783</td>
<td>0.36783</td>
<td>-0.3803E-09</td>
<td>0.36792</td>
<td>0.36792</td>
<td>0.4422E-09</td>
</tr>
<tr>
<td>t</td>
<td>u1(num)</td>
<td>u1(ex)</td>
<td>diff1</td>
<td>u2(num)</td>
<td>u2(ex)</td>
<td>diff2</td>
</tr>
<tr>
<td>-----</td>
<td>---------</td>
<td>--------</td>
<td>--------</td>
<td>---------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>2.00</td>
<td>0.13534</td>
<td>0.13534</td>
<td>0.2271E-10</td>
<td>0.13534</td>
<td>0.13534</td>
<td>0.2278E-10</td>
</tr>
<tr>
<td>3.00</td>
<td>0.04979</td>
<td>0.04979</td>
<td>0.1255E-10</td>
<td>0.04979</td>
<td>0.04979</td>
<td>0.1255E-10</td>
</tr>
<tr>
<td>4.00</td>
<td>0.01832</td>
<td>0.01832</td>
<td>0.6156E-11</td>
<td>0.01832</td>
<td>0.01832</td>
<td>0.6156E-11</td>
</tr>
<tr>
<td>5.00</td>
<td>0.00674</td>
<td>0.00674</td>
<td>0.2831E-11</td>
<td>0.00674</td>
<td>0.00674</td>
<td>0.2831E-11</td>
</tr>
</tbody>
</table>

rkc4b integrator

<table>
<thead>
<tr>
<th>t</th>
<th>u1(num)</th>
<th>u1(ex)</th>
<th>diff1</th>
<th>u2(num)</th>
<th>u2(ex)</th>
<th>diff2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.0000E+00</td>
<td>2.00000</td>
<td>2.00000</td>
<td>0.0000E+00</td>
</tr>
<tr>
<td>1.00</td>
<td>0.36783</td>
<td>0.36783</td>
<td>-0.1801E-07</td>
<td>0.36792</td>
<td>0.36792</td>
<td>0.2037E-07</td>
</tr>
<tr>
<td>2.00</td>
<td>0.13534</td>
<td>0.13534</td>
<td>0.1535E-08</td>
<td>0.13534</td>
<td>0.13534</td>
<td>0.1541E-08</td>
</tr>
<tr>
<td>3.00</td>
<td>0.04979</td>
<td>0.04979</td>
<td>0.7235E-08</td>
<td>0.04979</td>
<td>0.04979</td>
<td>0.7235E-08</td>
</tr>
<tr>
<td>4.00</td>
<td>0.01832</td>
<td>0.01832</td>
<td>0.5115E-08</td>
<td>0.01832</td>
<td>0.01832</td>
<td>0.5115E-08</td>
</tr>
<tr>
<td>5.00</td>
<td>0.00674</td>
<td>0.00674</td>
<td>0.1710E-07</td>
<td>0.00674</td>
<td>0.00674</td>
<td>0.1710E-07</td>
</tr>
</tbody>
</table>

rkf45a integrator

<table>
<thead>
<tr>
<th>t</th>
<th>u1(num)</th>
<th>u1(ex)</th>
<th>diff1</th>
<th>u2(num)</th>
<th>u2(ex)</th>
<th>diff2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.0000E+00</td>
<td>2.00000</td>
<td>2.00000</td>
<td>0.0000E+00</td>
</tr>
<tr>
<td>1.00</td>
<td>0.36783</td>
<td>0.36783</td>
<td>0.4425E-11</td>
<td>0.36792</td>
<td>0.36792</td>
<td>-0.4491E-11</td>
</tr>
<tr>
<td>t</td>
<td>u1(num)</td>
<td>u1(ex)</td>
<td>diff1</td>
<td>u2(num)</td>
<td>u2(ex)</td>
<td>diff2</td>
</tr>
<tr>
<td>-------</td>
<td>---------</td>
<td>--------</td>
<td>--------</td>
<td>---------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>0.00</td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.0000E+00</td>
<td>2.00000</td>
<td>2.00000</td>
<td>0.0000E+00</td>
</tr>
<tr>
<td>1.00</td>
<td>0.36783</td>
<td>0.36783</td>
<td>0.8641E-06</td>
<td>0.36792</td>
<td>0.36792</td>
<td>-0.8701E-06</td>
</tr>
<tr>
<td>2.00</td>
<td>0.13534</td>
<td>0.13534</td>
<td>-0.1452E-06</td>
<td>0.13534</td>
<td>0.13534</td>
<td>-0.1489E-06</td>
</tr>
<tr>
<td>3.00</td>
<td>0.04978</td>
<td>0.04979</td>
<td>-0.2150E-05</td>
<td>0.04978</td>
<td>0.04979</td>
<td>-0.2139E-05</td>
</tr>
<tr>
<td>4.00</td>
<td>0.01831</td>
<td>0.01832</td>
<td>-0.1685E-05</td>
<td>0.01831</td>
<td>0.01832</td>
<td>-0.1430E-05</td>
</tr>
<tr>
<td>5.00</td>
<td>0.00674</td>
<td>0.00674</td>
<td>-0.3822E-05</td>
<td>0.00674</td>
<td>0.00674</td>
<td>0.2111E-05</td>
</tr>
</tbody>
</table>

Generally, the accuracy of the numerical solution meets or exceeds the tolerances set in `intpar`.

### 3.5 Programming in Java

Again, since main Program 2.5.1 and interface routines 2.5.2 are unchanged in the 2x2 ODE problem, they are not listed here. `intpar`, `par`, `initial`, `derv`, and `fprint` are listed below:
/** This file is a member of the package mol */
package mol;

import mol.MOL;
import java.math.*;
import java.io.*;
import java.text.*;

public class DEF extends MOL implements ode2x2interface
{
    public DEF()
    {
        /* Integration parameters */
        this.intpar();

        /* Declare arrays */
u0=new double[SIZE];
u=new double[SIZE];
e=new double[SIZE];

        /* Problem parameters */
        this.par();

        /* Initial condition vector */
        this.initial();
    }

    public void intpar()
    {
        /* Function intpar sets the parameters to control the integration of the 2 x 2 ODE system */
        {
            /* Number of ODEs */
            neqn=2;

            /* Size of arrays in MOL library */
            SIZE=neqn+1;

            /* Number of output points */
            nout=6;

            /* Maximum number of steps in the interval t0 to tf */
    }
nsteps=100;

/* Initial, final values of the independent variable */
t0=0.0;
tf=1.0;

/* Error tolerances */
abserr=Math.pow(10.0,-5.0);
relerr=Math.pow(10.0,-5.0);

/* End of inpar */
}

public void initial()

/* Function initial sets the initial condition vector for
the 2 x 2 ODE problem */
{
    u0[1]=0.0E0;
    u0[2]=2.0E0;

    /* End of initial */
}

public void par()

/* Function par sets the parameters for the 2 x 2 ODE
problem */
{
    a=5.5;
    b=4.5;

    /* End of par */
}

public void deriv(double ut[], double t, double u[])

/* Function deriv computes the derivative vector of the
2 x 2 ODE problem */
{
    /* Problem parameters */
    par();
/* Derivative vector */

/* End of deriv */
}

public void fprint(PrintWriter f, int ncase, int neqn, double t, double u[])

/* Function fprint displays the numerical and exact solutions to the 2 x 2 ODE problem */
{
    /* Type variables */
    double ue1, ue2;
    double diff1, diff2;
    double e1, e2;

    /* Print a heading for the solution at t = 0 */
    if(t<=0.0)
    {
        /* Label for ODE integrator */
        switch(ncase)
        {
            /*Fixed step modified Euler */
            case 1:
                f.println("\n euler2a integrator\n");
                break;

            /* Variable step modified Euler */
            case 2:
                f.println("\n euler2b integrator\n");
                break;

            /* Fixed step classical fourth order RK */
            case 3:
                f.println("\n rkc4a integrator\n");
                break;

            /* Variable step classical fourth order RK */
            case 4:
                f.println("\n rkc4b integrator\n");
                break;
        }
    }
}
```c
/* Fixed step RK Fehlberg 45 */
case 5:
    f.println("\n rKF45a integrator\n");
    break;
/* Variable step RK Fehlberg 45 */
case 6:
    f.println("\n rKF45b integrator\n");
    break;
}
/* Heading */
f.println(" t u1(num) u1(ex) diff1*");
f.println(" u2(num) u2(ex) diff2*");
/* End of t = 0 heading */
}
/* Analytical solution */
ue1=Math.exp(-(a-b)*t)-Math.exp(-(a+b)*t);
ue2=Math.exp(-(a-b)*t)+Math.exp(-(a+b)*t);
/* Difference between exact and numerical solution vectors */
diff1=u[1]-ue1;
diff2=u[2]-ue2;
/* Display format for floating numbers */
DecimalFormat df1 = new DecimalFormat(" 0.00");
DecimalFormat df2 = new DecimalFormat("0.0000000");
/* Display the numerical and exact solutions, and their difference */
f.println("\n+t*df1.format(t)++"+df1.format(u[1])
    +"+df2.format(ue1)+"+df2.format(diff1));
f.println(" +"+df2.format(u[2])
    +"+df2.format(ue2)+"+df2.format(diff2));
/* End of fprint */
}
/* End of DEF */
}

Program 3.5.1
intpar, initial, par, derv, and fprint for the solution of Equations 1.6 and 1.16
```
The output from the preceding routines is as follows:

**euler2a integrator**

<table>
<thead>
<tr>
<th>t</th>
<th>u1(num)</th>
<th>u1(ex)</th>
<th>diff1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>0.0000000</td>
<td>0.000000</td>
<td>0.000000</td>
</tr>
<tr>
<td></td>
<td>2.0000000</td>
<td>2.000000</td>
<td>0.000000</td>
</tr>
<tr>
<td>1.00</td>
<td>0.3678394</td>
<td>0.367834</td>
<td>0.0000054</td>
</tr>
<tr>
<td></td>
<td>0.3679318</td>
<td>0.3679248</td>
<td>0.0000070</td>
</tr>
<tr>
<td>2.00</td>
<td>0.1353398</td>
<td>0.1353353</td>
<td>0.0000045</td>
</tr>
<tr>
<td></td>
<td>0.1353398</td>
<td>0.1353353</td>
<td>0.0000045</td>
</tr>
<tr>
<td>3.00</td>
<td>0.0497896</td>
<td>0.0497871</td>
<td>0.0000025</td>
</tr>
<tr>
<td></td>
<td>0.0497896</td>
<td>0.0497871</td>
<td>0.0000025</td>
</tr>
<tr>
<td>4.00</td>
<td>0.0183169</td>
<td>0.0183156</td>
<td>0.0000012</td>
</tr>
<tr>
<td></td>
<td>0.0183169</td>
<td>0.0183156</td>
<td>0.0000012</td>
</tr>
<tr>
<td>5.00</td>
<td>0.0067385</td>
<td>0.0067379</td>
<td>0.0000006</td>
</tr>
<tr>
<td></td>
<td>0.0067385</td>
<td>0.0067379</td>
<td>0.0000006</td>
</tr>
</tbody>
</table>

**euler2b integrator**

<table>
<thead>
<tr>
<th>t</th>
<th>u1(num)</th>
<th>u1(ex)</th>
<th>diff1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>0.0000000</td>
<td>0.000000</td>
<td>0.000000</td>
</tr>
<tr>
<td></td>
<td>2.0000000</td>
<td>2.000000</td>
<td>0.000000</td>
</tr>
<tr>
<td>1.00</td>
<td>0.3678395</td>
<td>0.367834</td>
<td>0.0000055</td>
</tr>
<tr>
<td></td>
<td>0.3679320</td>
<td>0.3679248</td>
<td>0.0000071</td>
</tr>
<tr>
<td>2.00</td>
<td>0.1353399</td>
<td>0.1353353</td>
<td>0.0000046</td>
</tr>
<tr>
<td></td>
<td>0.1353399</td>
<td>0.1353353</td>
<td>0.0000046</td>
</tr>
<tr>
<td>3.00</td>
<td>0.0497897</td>
<td>0.0497871</td>
<td>0.0000027</td>
</tr>
<tr>
<td></td>
<td>0.0497897</td>
<td>0.0497871</td>
<td>0.0000027</td>
</tr>
<tr>
<td></td>
<td>rkc4a integrator</td>
<td></td>
<td>rkc4b integrator</td>
</tr>
<tr>
<td>----</td>
<td>-----------------</td>
<td>----</td>
<td>-----------------</td>
</tr>
<tr>
<td></td>
<td>t</td>
<td>u1(num)</td>
<td>u1(ex)</td>
</tr>
<tr>
<td>4.00</td>
<td>0.0183176</td>
<td>0.0183156</td>
<td>0.0000019</td>
</tr>
<tr>
<td>5.00</td>
<td>0.0067398</td>
<td>0.0067379</td>
<td>0.0000018</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>0.0000000</td>
<td>0.0000000</td>
<td>0.0000000</td>
</tr>
<tr>
<td>1.00</td>
<td>0.3678340</td>
<td>0.3678340</td>
<td>-0.0000000</td>
</tr>
<tr>
<td>2.00</td>
<td>0.1353353</td>
<td>0.1353353</td>
<td>0.0000000</td>
</tr>
<tr>
<td>3.00</td>
<td>0.0497871</td>
<td>0.0497871</td>
<td>0.0000000</td>
</tr>
<tr>
<td>4.00</td>
<td>0.0183156</td>
<td>0.0183156</td>
<td>0.0000000</td>
</tr>
<tr>
<td>5.00</td>
<td>0.0067379</td>
<td>0.0067379</td>
<td>0.0000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>0.0000000</td>
<td>0.0000000</td>
<td>0.0000000</td>
</tr>
<tr>
<td>1.00</td>
<td>0.3678340</td>
<td>0.3678340</td>
<td>-0.0000000</td>
</tr>
<tr>
<td>2.00</td>
<td>0.1353353</td>
<td>0.1353353</td>
<td>0.0000000</td>
</tr>
</tbody>
</table>

Copyright 2004 by Chapman & Hall/CRC
rkf45a integrator

<table>
<thead>
<tr>
<th>t</th>
<th>u1(num)</th>
<th>u1(ex)</th>
<th>diff1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.0000000</td>
<td>0.0000000</td>
<td>0.0000000</td>
</tr>
<tr>
<td>2.00</td>
<td>2.0000000</td>
<td>2.0000000</td>
<td>0.0000000</td>
</tr>
<tr>
<td>1.00</td>
<td>0.3678340</td>
<td>0.3678340</td>
<td>0.0000000</td>
</tr>
<tr>
<td></td>
<td>0.3679248</td>
<td>0.3679248</td>
<td>-0.0000000</td>
</tr>
<tr>
<td>2.00</td>
<td>0.1353353</td>
<td>0.1353353</td>
<td>-0.0000000</td>
</tr>
<tr>
<td></td>
<td>0.1353353</td>
<td>0.1353353</td>
<td>-0.0000000</td>
</tr>
<tr>
<td>3.00</td>
<td>0.0497871</td>
<td>0.0497871</td>
<td>-0.0000000</td>
</tr>
<tr>
<td></td>
<td>0.0497871</td>
<td>0.0497871</td>
<td>-0.0000000</td>
</tr>
<tr>
<td>4.00</td>
<td>0.0183156</td>
<td>0.0183156</td>
<td>-0.0000000</td>
</tr>
<tr>
<td></td>
<td>0.0183156</td>
<td>0.0183156</td>
<td>-0.0000000</td>
</tr>
<tr>
<td>5.00</td>
<td>0.0067380</td>
<td>0.0067379</td>
<td>-0.0000000</td>
</tr>
<tr>
<td></td>
<td>0.0067380</td>
<td>0.0067379</td>
<td>-0.0000000</td>
</tr>
</tbody>
</table>

rkf45b integrator

<table>
<thead>
<tr>
<th>t</th>
<th>u1(num)</th>
<th>u1(ex)</th>
<th>diff1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.0000000</td>
<td>0.0000000</td>
<td>0.0000000</td>
</tr>
<tr>
<td>2.00</td>
<td>2.0000000</td>
<td>2.0000000</td>
<td>0.0000000</td>
</tr>
<tr>
<td>1.00</td>
<td>0.3678349</td>
<td>0.3678340</td>
<td>0.0000009</td>
</tr>
<tr>
<td></td>
<td>0.3679240</td>
<td>0.3679248</td>
<td>-0.0000009</td>
</tr>
</tbody>
</table>
Generally, the accuracy of the numerical solution meets or exceeds the toler-
ances set in \textit{intpar}.

3.6 Programming in Maple

Since main Program 3.6.1 (and subordinate routines) accesses specific files by \textit{read} statements, it is listed first:

\begin{verbatim}
> restart:
> read "c:\odelib\maple\ode2x2\ode2x2.txt";
> ode2x2();
\end{verbatim}

Program 3.6.1
Maple main program \texttt{ode2x2.mws} for the numerical integration of Equations 1.6 and 1.16

\begin{verbatim}
ode2x2:=proc()
#
# Main program ode2x2 computes the numerical
# solution to the 2 x 2 ODE system by one of
# six integrators
#
# Type variables
  global neqn, nout, nsteps, t0, tf, abserr, relerr:
  local u0, u, tp, ncase, i, j:
#
# Step through six integrators
  for ncase from 1 to 6 do

\end{verbatim}
# Integration parameters
read "c:\odelib\maple\ode2x2\intpar.txt":
intpar():
#
# Size arrays
u0:=array(1..neqn): u:=array(1..neqn):
#
# Initial condition vector
read "c:\odelib\maple\ode2x2\inital.txt":
inital(n,t0,u0):
#
# Output interval
tp:=tf-t0:
#
# Compute solution at nout output points
for j from 1 to nout do
#
# Print current solution
read "c:\odelib\maple\ode2x2\fprint.txt":
fprint(ncase,neqn,t0,u0):
#
# Fixed step modified Euler integrator
if (ncase = 1) then
  read "c:\odelib\maple\ode2x2\euler2a.txt":
euler2a(neqn,t0,tf,u0,nsteps,u):
end if:
#
# Variable step modified Euler integrator
if (ncase = 2) then
  read "c:\odelib\maple\ode2x2\euler2b.txt":
euler2b(neqn,t0,tf,u0,nsteps,abserr,relerr,u):
end if:
#
# Fixed step classical fourth order RK integrator
if (ncase = 3) then
  read "c:\odelib\maple\ode2x2\rkc4a.txt":
rkc4a(neqn,t0,tf,u0,nsteps,u):
end if:
#
# Variable step classical fourth order RK integrator
if (ncase = 4) then
  read "c:\odelib\maple\ode2x2\rkc4b.txt":
rkc4b(neqn,t0,tf,u0,nsteps,abserr,relerr,u):
end if:
# Fixed step RK Fehlberg (RKF45) integrator
if (ncase = 5) then
    read "c:\odelib\maple\ode2x2\rkf45a.txt":
rkf45a(neqn,t0,tf,u0,nsteps,u): end if:
#
# Variable step RK Fehlberg (RKF45) integrator
if (ncase = 6) then
    read "c:\odelib\maple\ode2x2\rkf45b.txt":
rkf45b(neqn,t0,tf,u0,nsteps,abserr,relerr,u): end if:
#
# Advance solution
t0:=tf:
tf:=tf+tp:
for i from 1 to neqn do
    u0[i]:=u[i]: end do:
#
# Next output
end do:
#
# Next integrator
end do:
#
# End of ode2x2.txt
end:

Program 3.6.2
Maple main program ode2x2.txt for the numerical integration of Equations 1.6 and 1.16

Note the reference to specific files by read statements, e.g.,

# Initial condition vector
read "c:\odelib\maple\ode2x2\initial.txt":
initial(neqn,t0,u0):

intpar:=proc()
#
# Function intpar sets the parameters to control the
# integration of the 2 x 2 ODE problem
# Type variables
  global neqn, nout, nsteps, t0, tf, abserr, relerr:
#
# Number of first order ODEs
  neqn:=2:
#
# Number of output points
  nout:=6:
#
# Maximum number of steps in the interval t0 to tf
  nsteps:=100:
#
# Initial, final values of independent variable
  t0:=0.0:
  tf:=1.0:
#
# Error tolerances
  abserr:=1.0e-05:
  relerr:=1.0e-05:
#
# End of intpar
  end:

inital:=proc(neqn,t,u0)
#
# Procedure inital sets the initial condition vector
# for the 2 x 2 ODE problem
#
  u0[1]:=0:
  u0[2]:=2:
#
# End of inital
  end:

derv:=proc(neqn,t,u,ut)
#
# Procedure deriv computes the derivative vector
# of the 2 x 2 ODE problem
#
# Type variables
  global a, b:
#
# Problem parameters
a:=5.5:
b:=4.5:
#
# Derivative vector
ut[1]:=-a*u[1]+b*u[2]:
ut[2]:= b*u[1]-a*u[2]:
#
# End of derv
end:

fprint:=proc(ncase,neqn,t,u)
#
# Procedure fprint displays the numerical and
# exact solutions to the 2 x 2 ODE problem
#
# Type variables
  global a, b:
  local e1, e2, ue, diff, i:
#
# Define arrays
  ue:=array(1..neqn): diff:=array(1..neqn):
#
# Print a heading for the solution at t = 0
  if (t <= 0.0) then
    # Label for ODE integrator
    #
    # Fixed step modified Euler
    if (ncase = 1) then
      printf(`
      euler2a integrator
      `);
    #
    # Variable step modified Euler
    elif (ncase = 2) then
      printf(`
      euler2b integrator
      `);
    #
    # Fixed step classical fourth order RK
    elif (ncase = 3) then
      printf(`
      rkc4a integrator
      `);
    #
    # Variable step classical fourth order RK
    elif (ncase = 4) then
      printf(`
      rkc4b integrator
      `);
  #
Program 3.6.3

*intpar, initial, derv, and fprint* for the solution of Equations 1.6 and 1.16

The output from the preceding routines is as follows:
<table>
<thead>
<tr>
<th>t</th>
<th>u1</th>
<th>u2</th>
<th>u1-ue1</th>
<th>u2-ue2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.00000</td>
<td>2.00000</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>1.00</td>
<td>0.36784</td>
<td>0.36793</td>
<td>0.0001</td>
<td>0.0001</td>
</tr>
<tr>
<td>2.00</td>
<td>0.13534</td>
<td>0.13534</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>3.00</td>
<td>0.04979</td>
<td>0.04979</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>4.00</td>
<td>0.01832</td>
<td>0.01832</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>5.00</td>
<td>0.00674</td>
<td>0.00674</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>t</th>
<th>u1</th>
<th>u2</th>
<th>u1-ue1</th>
<th>u2-ue2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.00000</td>
<td>2.00000</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>1.00</td>
<td>0.36783</td>
<td>0.36792</td>
<td>-0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>2.00</td>
<td>0.13534</td>
<td>0.13534</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>3.00</td>
<td>0.04979</td>
<td>0.04979</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>4.00</td>
<td>0.01832</td>
<td>0.01832</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>5.00</td>
<td>0.00674</td>
<td>0.00674</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>t</th>
<th>u1</th>
<th>u2</th>
<th>u1-ue1</th>
<th>u2-ue2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.00000</td>
<td>2.00000</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>1.00</td>
<td>0.36783</td>
<td>0.36792</td>
<td>-0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>2.00</td>
<td>0.13534</td>
<td>0.13534</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>3.00</td>
<td>0.04979</td>
<td>0.04979</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>4.00</td>
<td>0.01832</td>
<td>0.01832</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>5.00</td>
<td>0.00674</td>
<td>0.00674</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
</tbody>
</table>
Generally, the accuracy of the numerical solution meets or exceeds the tolerances set in `intpar`.

This completes the discussion of the 2x2 ODE problem programmed in the six languages. Basically, what we have considered is the use of the library integrations for the solution of nxn systems of ODEs (as illustrated by the solution of the 2x2 system).

We again point out that the preceding numerical solutions are for \( a = 5.5, \ b = 4.5 \) corresponding to the nonstiff case \( \lambda_1 = -1, \lambda_2 = -10 \). As expected, this problem can be handled efficiently and with good accuracy by the six nonstiff integrators (stability is not a problem). However, for the stiff case \( a = 500,000.5, b = 499,999.5 \) listed after Equations 1.54, a stiff integrator should be used to efficiently handle the problem of stability. This requirement (for a stiff or implicit integrator) is discussed in some detail in Appendix C.

Thus, we emphasize that the library integrators discussed previously have limitations (as do all numerical algorithms). They are therefore intended to serve as a starting point, and to demonstrate some basic concepts and approaches. But success in solving any particular problem cannot be guaranteed in advance, and generally some experimentation with the choice of integrators and parameters (such as error tolerances) is required to arrive at a solution with acceptable accuracy and computational effort.
For example, we offer the suggestion that for a new ODE problem, a nonstiff (explicit) integrator should be tried first. Our experience has indicated that a broad spectrum of problems can be handled in this way. If the calculations appear to be excessive, possibly signaling stiffness, then a switch to a stiff integrator is a logical next step.

We now consider two problems in PDEs in the next two chapters. We shall see that the preceding techniques for ODEs can also be applied to PDEs.
4

Solution of a Linear PDE

The preceding methods, e.g., the (1, 2), (2, 4) and (4, 5) RK pairs, can be applied to PDEs by using the numerical method of lines. Briefly, the spatial (boundary value) derivatives in the PDEs are replaced with algebraic approximations, typically finite differences. The resulting equations are ODEs in the initial value variable, previously designated \( t \), so that we have an initial value problem in ODEs. This is exactly the problem we considered in Chapters 1 through 3.

We now illustrate this approach to PDEs through two examples: (1) a linear PDE problem in this chapter and (2) a nonlinear PDE problem in the next chapter. The linear PDE problem is just Fourier’s second law, which is also called the heat conduction equation:

\[
\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} \tag{4.1}
\]

Briefly, here is some terminology and associated mathematical conditions for Equation 4.1:

- \( u \) is the dependent variable. Although Equation 4.1 typically describes the temperature distribution in a solid, and therefore \( T \) (for temperature) would seem a more logical name for the dependent variable, by convention, at least in much of the mathematical literature, \( u \) is used for a PDE dependent variable.
- \( x \) and \( t \) are the independent variables. There are two types of such independent variables:
  - \( x \) is a spatial or boundary value (BV) variable. As these names imply, \( x \) defines a spatial scale, i.e., a distance. Also, our physical world is three dimensional, so Equation 4.1 is a one-dimensional (1D) approximation. Such an approximation might seem unrealistic (since the world is three dimensional), but there are often circumstances under which a 1D (or 2D) approximation is quite adequate; we do not go into
details here, but the use of a lower-dimensional approximation is quite common.

— $t$ is an initial value (IV) variable, generally time.

- The features that distinguish IV from BV variables are considered subsequently.

- Equation 4.1 is linear since the dependent variable $u$ is to the first power. In other words, Equation 4.1 is first degree.

- Equation 4.1 might appear to not be linear or first degree because of the derivative $\partial^2 u / \partial x^2$. However, this is a second-order derivative. In other words, Equation 4.1 is first degree (linear) and second order (in $x$). It is easy to confuse order and degree!

- Equation 4.1 is a parabolic PDE. This geometric classification is one of three possibilities: elliptic, parabolic, or hyperbolic. We do not get into the details of this classification of PDEs, but it is quite important in understanding the properties of the PDE solution and in the selection of numerical methods for the calculation of numerical solutions. More details of the geometric classification of PDEs, and some of the other nomenclature we are considering here, can be found in many books (e.g., Wouver et al., chap. 1).

- The solution to Equation 4.1 is the dependent variable, $u$, as a function of the independent variables, $x$ and $t$, that is $u(x, t)$. As in the case of the ODEs discussed in Chapters 1 through 3, the solution to a PDE might be (1) exact or analytical or (2) numerical.

- Since Equation 4.1 is first order in $t$, it requires one initial condition in $t$, which we take as

$$u(x, 0) = u_0(x) \quad (4.2)$$

We can note the following points about this initial condition:

— The initial value independent variable is specified at only one value, in this case $t = 0$. Even if we had a system of PDEs, each requiring, an initial condition, all of the initial conditions would be specified at the same value of the initial value independent variable. In other words, we can think of this single value as a starting point for the solution; hence the name “initial.”

— The initial condition can be a function of the other PDE independent variables, in this case $u_0(x)$. For Equation 4.1, we take $u_0(x) = \sin(\pi x)$.

— For a PDE which is $n$th order in the initial value variable, $n$ initial conditions are required. In other words, a differential equation requires a number of initial conditions equal to the order of the highest-order derivative in the initial value independent variable. In the case of Equation 4.1, $n = 1$. Also, all of the ODEs considered in Chapters 1 through 3 were first order in $t$, and therefore each ODE required only one initial condition.
Since Equation 4.1 is second order in $x$, it requires two boundary conditions. We will take these to be

\begin{align*}
u(0, t) &= 0 \quad (4.3) \\
u(1, t) &= 0 \quad (4.4)
\end{align*}

We can note the following about Equations 4.3 and 4.4:

- The boundary value independent variable, $x$, is specified at two different values, $x = 0$ and $x = 1$. In other words, boundary conditions are specified at two more values of the boundary value independent variable, in contrast to initial conditions, which are specified at the same value of the independent variable.

- The boundary conditions could be functions of the other independent variables, e.g.,

\begin{align*}
u(0, t) &= f_1(t), \quad u(1, t) = f_2(t) \quad (4.5)
\end{align*}

For Equations 4.3 and 4.4, we have taken $f_1(t) = f_2(t) = 0$.

- As the name “boundary condition” implies, the values of the boundary value independent variable at which the boundary conditions are specified are often boundaries of a physical system.

- Whereas the initial value problem is open-ended in the sense that the range of the initial value variable (e.g., $t$) is unlimited and typically is the semi-infinite interval, $0 \leq t \leq \infty$, the range of the boundary value variable is often finite; in the case of Equations 4.3 and 4.4, $0 \leq x \leq 1$. However, the limits of the boundary value independent variable can be semi-infinite, e.g., $0 \leq x \leq \infty$ or fully infinite $-\infty \leq x \leq \infty$. This implies a physical system of infinite dimensions, which may actually be a good approximation.

- PDEs are often expressed in an alternative subscript notation. For example, we can express Equation 4.1 as

\begin{align*}
u_t &= u_{xx} \quad (4.6)
\end{align*}

where

- A subscript denotes a partial derivative with respect to an independent variable, e.g., $u_t = \partial u/\partial t$.

- The subscript is repeated a number of times equal to the order of the derivative, e.g., $u_{xx} = \partial^2 u/\partial x^2$.

This subscript notation has the advantage of simplicity in expressing the PDE and in writing the associated computer code (the use of conventional partial derivatives is avoided).

Equations 4.1 (or 4.6), 4.2, 4.3, and 4.4 constitute a complete, well-posed PDE problem and we can proceed to the solution, $u(x, t)$. To reiterate, we seek
as the solution of a system of differential equations the dependent variables as functions of the independent variables. We have already observed this in the case of the two preceding ODE problems, i.e., (1) Equation 1.5 for \( y(t) \) is the analytical solution to Equation 1.3 and (2) Equations 1.17 for \( [y_1(t) \ y_2(t)]^T \) are the analytical solutions to Equations 1.6. We now consider the analytical and numerical solutions of Equations 4.1 to 4.4.

The analytical solution is

\[
u(x, t) = \sin(\pi x) e^{-\pi^2 t}
\]  

(4.7)

Equation 4.7 can easily be verified as the solution to Equations 4.1 to 4.4:

<table>
<thead>
<tr>
<th>Equation</th>
<th>Term</th>
<th>Equation 4.7</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>( \frac{\partial u}{\partial t} )</td>
<td>( -\pi^2 \sin(\pi x) e^{-\pi^2 t} )</td>
</tr>
<tr>
<td></td>
<td>( \frac{\partial^2 u}{\partial x^2} )</td>
<td>( \pi^2 \sin(\pi x) e^{-\pi^2 t} )</td>
</tr>
<tr>
<td></td>
<td>=0</td>
<td>=0</td>
</tr>
<tr>
<td>4.2</td>
<td>( u(x, 0) )</td>
<td>( \sin(\pi x) )</td>
</tr>
<tr>
<td>4.3</td>
<td>( u(0, t) )</td>
<td>0</td>
</tr>
<tr>
<td>4.4</td>
<td>( u(1, t) )</td>
<td>0</td>
</tr>
</tbody>
</table>

Thus, we can use Equation 4.7 to evaluate a numerical solution to Equations 4.1 to 4.4; that is, we can compute the exact error in the numerical solution using Equation 4.7.

Before we can proceed to the calculation of a numerical solution to Equations 4.1 to 4.4, we first have to develop a method to program PDEs, and in particular, Equation 4.1. We actually faced this problem with the ODEs of Chapter 1; i.e., there is no way that a computer can accept ODEs directly (since computers can basically only do high-speed arithmetic and store the results).

In the case of ODEs, we adopted RK algorithms, which reduce the solution of ODEs to high-speed arithmetic. In other words, calculation of ODE numerical solutions required only the numerical evaluation of derivatives in the routine \( \text{derv} \), and stepping along the solution using the RK stepping formulas such as Equation 1.19 for the Euler method—note that Equation 1.19 required only the numerical value of the derivative \( dy_i/dt \) (from \( \text{derv} \)), followed by multiplication by \( h \) and addition to \( y_i \), that is, just high-speed arithmetic! In general, all of the RK methods, (1, 2), (2, 3), (2, 4), and (4, 5), ultimately require only the numerical calculation of derivatives and arithmetic.

Thus, we first must develop a method for solving PDEs using only what the computer can do. Certainly a computer is not well suited for understanding partial derivatives and generating a solution like Equation 4.7 (admittedly, computer algebra systems such as Maple can produce analytical solutions.
to a limited class of PDE problems, including Equation 4.7, but we want to develop a general numerical method for PDEs that can be applied to large, complex systems of PDEs for which computer algebra systems are unable to produce analytical solutions.

To this end, we first replace the derivative $\frac{\partial^2 u}{\partial x^2}$ in Equation 4.1 ($u_{xx}$ in Equation 4.6) with a finite difference (FD) approximation:

$$\frac{\partial^2 u}{\partial x^2} \mid_{x=i} \simeq \frac{u_{i+1} - 2u_{i} + u_{i-1}}{\Delta x^2} + O(\Delta x^2), \quad i = 1, 2, \ldots, n \quad (4.8)$$

We can note the following points about Equation 4.8:

- A spatial grid in $x$ is defined over the interval $0 \leq x \leq 1$. Along this grid, the value of a particular value of $x$ is designated with the index $i$. Thus, $i = 1$ corresponds to $x = 0$ and $i = n$ corresponds to $x = 1$. The spacing between adjacent grid points is $\Delta x$. In other words, $\Delta x = x_i - x_{i-1} = x_{i+1} - x_i$ or $\Delta x = (1 - 0)/(n - 1)$.

- At a particular point along the grid, $i$, the second-order derivative $\frac{\partial^2 u}{\partial x^2} \mid_{x=i}$ is approximated by the RHS of Equation 4.8, i.e., a three point FD approximation since it involves the value of $u$ at three points, $u_{i+1}, u_i,$ and $u_{i-1}$.

- Most importantly, note that the RHS of Equation 4.8 does not have a derivative; rather, it is entirely algebraic, and can therefore be easily computed (since a computer does high-speed arithmetic). In other words, we have replaced the derivative $\frac{\partial^2 u}{\partial x^2}$, which a computer cannot handle, with an algebraic approximation, which a computer can easily evaluate.

- $O(\Delta x^2)$ indicates that the approximation of Equation 4.8 is second order in $x$, i.e., the truncation error of the approximation is proportional to $\Delta x^2$. We can therefore expect to compute more accurate values of $\frac{\partial^2 u}{\partial x^2}$ from Equation 4.8 as we use more grid points (larger $n$, and therefore smaller $\Delta x$).

If the FD approximation of Equation 4.8 is substituted in Equation 4.1, we have

$$\frac{du_i}{dt} \simeq \frac{u_{i+1} - 2u_{i} + u_{i-1}}{\Delta x^2}, \quad i = 1, 2, \ldots, n \quad (4.9)$$

We can note the following points about Equation 4.9:

- It has only one independent variable, $t$. Thus, it is an ODE!

- Actually, Equation 4.9 represents a system of $n$ ODEs for $i = 1, 2, \ldots, n$. Furthermore, they are initial value ODEs since Equation 4.2 provides an initial condition, that is,

$$u(x_i, 0) = u_i(t) = \sin(\pi x_i), \quad i = 1, 2, \ldots, n \quad (4.10)$$

Thus, we now have a system of initial value ODEs, Equation 4.9, in place of a PDE, Equation 4.1.
• Since Chapters 1 to 3 are devoted to the solution of systems of initial value ODEs, we now have methods and routines for the solution of Equations 4.9, and therefore, we can compute a numerical to PDE Equation 4.1 for comparison with the analytical solution of Equation 4.8.

• Basically all that is required is to step along the spatial grid and integrate the corresponding ODEs from Equations 4.9. For \( i = 1 \), we can apply BC (Equation 4.2) as

\[
 u(x_1, t) = u_1(t) = 0, \quad \frac{du_1}{dt} = 0
\]  

(4.11)

Similarly, for \( i = n \), BC (Equation 4.3) becomes

\[
 u(x_n, t) = u_n(t) = 0, \quad \frac{du_n}{dt} = 0
\]  

(4.12)

Equations 4.11 and 4.12 merely express the fact that if the boundary values are constant \( (u_1(t) = 0, u_n(t) = 0) \), their derivatives are zero \( (du_1/dt = 0, du_n/dt = 0) \).

To conclude this brief discussion of the approximation of PDEs as systems of ODEs (termed the method of lines), we point out that FD approximations of derivatives, such as Equation 4.8, are well developed. FD approximations of any order (e.g., \( O(\Delta x^2) \)) for derivatives of any order (e.g., \( \partial^2 u/\partial x^2 \)) are readily available.\(^2\)\(^\text{--}^4\) use an \( O(\Delta x^4) \) approximation of \( \partial^2 u/\partial x^2 \) as discussed in Appendix D.

We now proceed to the calculation of a numerical solution to Equations 4.1 to 4.4 using the six languages, starting with MATLAB. In each case, we use Equation 4.7 to evaluate the error in the numerical solution.

### 4.1 Programming in MATLAB

A main program for the solution of the linear PDE problems, Equations 4.1 to 4.4, is the same as Program 3.1.1 and therefore is not listed here. *intpar, initial, derv, and fprintf* are listed below followed by some discussion for each.

```matlab
function [neqn,nout,nsteps,t0,tf,abserr,relerr]=intpar
% Function intpar sets the parameters to control the
% integration of the linear PDE problem
% Number of first order ODEs
  neqn=21;
% %
```

Copyright 2004 by Chapman & Hall/CRC
% Number of output points
nout=6;
%
% Maximum number of steps in the interval t0 to tf
nsteps=100;
%
% Initial, final values of independent variable
t0=0.0;
tf=0.2;
%
% Error tolerances
abserr=1.0e-05;
relerr=1.0e-05;

Program 4.1.1
intpar for the solution of Equations 4.1 to 4.4

We can note the following points about intpar:

• The number of ODEs is 21:

% % Number of first order ODEs
  neqn=21;

In other words, we have defined a spatial grid with 21 points in using Equations 4.9. At this point, this selection of 21 is arbitrary and we will have to assess whether this is an adequate number of points, that is, whether it gives acceptable accuracy. Thus, we have an additional consideration that we did not have with ODEs, the spatial discretization error as set by the number of grid points.

• The output interval is set to 0.2:

% % Initial, final values of independent variable
  t0=0.0;
tf=0.2;

This is different from the output interval used for the preceding ODE problems (tf = 1.0). The necessity to use a smaller output interval is suggested by the analytical solution, Equation 4.7, \( u(x, t) = \sin(\pi x) e^{-\pi^2 t} \). The rate of change of the solution with \( t \) is determined by the exponential \( e^{-\pi^2 t} \). If the output interval is \( tf = 0.2 \), the exponential changes only modestly (by \( e^{-\pi^2 (0.2)} \) at each output) and the change in the solution with \( t \) can therefore be observed. If \( tf = 1.0 \) is used, the solution would decay to a small value at the end of one interval (it would change to \( e^{-\pi^2 (1)} \) of its initial value). This raises an important point; when selecting the output...
interval \( tf \) in \textit{intpar}, we must consider the \textit{timescale} of the problem. If we underestimate the timescale (make the total time interval too small), we will miss a significant part of the solution since the solution will not be completely computed (\( tf = 0.01 \) would do this with \( nout = 6 \) for Equations 4.1 to 4.4); if we overestimate the timescale (make the total time interval too large), we will again miss a significant part of the solution since the solution will essentially be completely computed within the first output interval (\( tf = 1.0 \) would do this with \( nout = 6 \)). Thus, we must be careful in specifying the output interval \( tf \) and the number of output points \( nout \) so as to achieve reasonable resolution of the solution in time. In other words, some \textit{tuning of the integration parameters is required}; using values that are not well thought out will cause the integrator to fail (as we shall observe when we use the \textit{intpar} in Program 4.1.1). In fact, \textit{a poor choice of the integration parameters is the single most common cause of integrator failures}.

The rest of \textit{intpar} is essentially self-explanatory (it follows from the previous discussion of the two ODE problems). \textit{inital} is listed below:

```
function [u]=inital(neqn,t)
  % Function inital sets the initial condition vector 
  % of the linear PDE problem 
  % Problem parameters 
  xl=0.0;
  xu=1.0;
  % Initial condition 
  for i=1:neqn
    x=xl+(i-1)/(neqn-1)*(xu-xl);
    u(i)=sin(pi*x);
  end
```

\textbf{Program 4.1.2}

\textit{inital} for the solution of Equations 4.1 to 4.4

\textit{inital} is an implementation of IC Equation 4.10. We can note the following points:

- The initial condition function, \( u(x_i, 0) = \sin(\pi x_i) \), is computed over the grid in \( x \) using the \textit{for} loop.
% Initial condition
for i=1:neqn
    x=xl+(i-1)/(neqn-1)*(xu-xl);
    u(i)=sin(pi*x);
end

Note that $x$ covers the interval $0 \leq x \leq 1$.

- $\pi$ is a predefined constant in MATLAB so it does not have to be assigned a numerical value (this is not the case in most of the other programming languages).

Thus, the 21 starting (initial) values of $u(x, 0)$ have been assigned and we can now consider the 21 ODEs from Equations 4.9, which are programmed in `derv`:

```matlab
function [ut]=derv(neqn,t,u)
    % Function derv computes the derivative vector
    % of the linear PDE problem
    % Problem parameters
    xl=0.0;
    xu=1.0;
    % BC at x = 0
    ut(1)=0.0;
    % BC at x = 1
    ut(neqn)=0.0;
    % Interior points
    dx=(xu-xl)/(neqn-1);
    dxs=dx*dx;
    for i=2:neqn-1
        ut(i)=(u(i+1)-2.0*u(i)+u(i-1))/dxs;
    end

Program 4.1.3
derv for the solution of Equations 4.1 to 4.4

We can note the following points about `derv`:

- The boundary values of $x$ are first set:

```
ODEs 4.11 and 4.12 are then programmed (here we make good use of the subscript notation of Equation 4.6 for partial derivatives, i.e., \( \frac{\partial u}{\partial t} = \partial_t \leftrightarrow u_t(i) \)):

% BC at \( x = 0 \)
\( u_t(1) = 0.0; \)
% BC at \( x = 1 \)
\( u_t(neqn) = 0.0; \)

The remaining ODEs for \( i = 2, 3, \ldots, neqn - 1 \) are then programmed in a for loop according to Equation 4.9:

% Interior points
\( dx = (xu-xl)/(neqn-1); \)
\( dxs = dx*dy; \)
for \( i=2:neqn-1 \)
\( u_t(i) = (u(i+1) - 2.0*u(i) + u(i-1))/dxs; \)
end

At the end of derv, all 21 ODEs of Equation 4.9 are programmed. These 21 derivatives \( (u_t(i)) \) are then returned to one of the ODE integrators, euler2a to rkf45b, to take the next step along the solution. In this way, the complete solution to Equations 4.1 to 4.4 is computed through \( t \) and \( x \), that is, we have \( u(x, t) \) in numerical form.

This numerical solution is displayed by fprint:

```matlab
function [out]=fprint(ncase,neqn,t,u)

% Function fprint displays the numerical and exact solutions to the linear PDE problem

% Declare global variables
global nsteps;

% Return current value of independent variable
% (MATLAB requires at least one return argument)
out=t;

% Problem parameters
xl=0.0;
xu=1.0;
```

Copyright 2004 by Chapman & Hall/CRC
% Print a heading for the solution at t = 0
if(t<=0.0)
%
% Label for ODE integrator
%
% Fixed step modified Euler
if(ncase==1)
    fprintf('

 euler2a integrator\n\n');
%
% Variable step modified Euler
elseif(ncase==2)
    fprintf('

 euler2b integrator\n\n');
%
% Fixed step classical fourth order RK
elseif(ncase==3)
    fprintf('

 rkc4a integrator\n\n');
%
% Variable step classical fourth order RK
elseif(ncase==4)
    fprintf('

 rkc4b integrator\n\n');
%
% Fixed step RK Fehlberg 45
elseif(ncase==5)
    fprintf('

 rkf45a integrator\n\n');
%
% Variable step RK Fehlberg 45
elseif(ncase==6)
    fprintf('

 rkf45b integrator\n\n');
end
%
% Heading
fprintf(' ncase = %2d neqn = %2d
 nsteps = %3d \n\n',ncase,neqn,nsteps);
fprintf(' t u(num) u(exact) diff\n
');
%
% End of t = 0 heading
end
%
% Numerical and analytical solution output
%
% Midpoint value of x
x=(xu-xl)/2.0;
%
% Analytical solution at midpoint
ue=exp(-pi*pi*t)*sin(pi*x);
% Grid index of midpoint
im=round((neqn+1)/2);
%
% Display the numerical and exact solutions,
% and their difference
fprintf('%5.2f %11.6f %11.6f %13.4e
',t,u(im),ue,
    u(im)-ue);

Program 4.1.4
fprint for the solution of Equations 4.1 to 4.4

fprint has the same function as the previous fprint routines for the two ODE problems, i.e.,

- Labeling of the ODE integrator
- Calculation of the analytical or exact solution (ue from Equation 4.7)
- Display of the numerical and analytical solutions, and their difference (u(im) – ue) (note that this comparison is made at the midpoint value x = 1/2)

This completes the programming of Equations 4.1 to 4.4 (or Equations 4.9). Execution of this program produced some interesting and unexpected results. First, the solution was unstable. This instability was apparent since the displayed numbers either (1) overflowed the formats of the fprintf statement in fprint or (2) produced NaN (not a number).

Thus, the stability analysis of Chapter 1 becomes quite relevant. Recall in Section 1.7 that the explicit RK methods discussed in Chapter 1 have a stability limit. For example, for the Euler method, |λh| < 2 set the maximum value of h for which an Euler integration remains stable. In the present case of the linear PDE approximated by Equation 4.9, this stability limit has apparently been reached (or exceeded).

To test this idea, we executed the program again with nsteps in intpar increased from 100 to 250 (so that h could be reduced by a factor of 100/250). This reduction in nsteps was, in fact, enough to stabilize the solution, which is listed below:

euler2a integrator

ncase =  1    neqn =  21    nsteps = 250

<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.0000e+00</td>
</tr>
<tr>
<td>0.20</td>
<td>0.139478</td>
<td>0.138911</td>
<td>5.6734e-04</td>
</tr>
<tr>
<td>0.40</td>
<td>0.019454</td>
<td>0.019296</td>
<td>1.5794e-04</td>
</tr>
</tbody>
</table>
### euler2b integrator

**ncase = 2  neqn = 21  nsteps = 250**

<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.00000000</td>
<td>1.00000000</td>
<td>0.0000e+00</td>
</tr>
<tr>
<td>0.20</td>
<td>0.139530</td>
<td>0.138911</td>
<td>6.1893e-04</td>
</tr>
<tr>
<td>0.40</td>
<td>0.019464</td>
<td>0.019296</td>
<td>1.6763e-04</td>
</tr>
<tr>
<td>0.60</td>
<td>0.002719</td>
<td>0.002680</td>
<td>3.8243e-05</td>
</tr>
<tr>
<td>0.80</td>
<td>0.000382</td>
<td>0.000372</td>
<td>9.2987e-06</td>
</tr>
<tr>
<td>1.00</td>
<td>0.000059</td>
<td>0.000052</td>
<td>7.0554e-06</td>
</tr>
</tbody>
</table>

### rkc4a integrator

**ncase = 3  neqn = 21  nsteps = 250**

<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.00000000</td>
<td>1.00000000</td>
<td>0.0000e+00</td>
</tr>
<tr>
<td>0.20</td>
<td>0.139476</td>
<td>0.138911</td>
<td>5.6448e-04</td>
</tr>
<tr>
<td>0.40</td>
<td>0.019453</td>
<td>0.019296</td>
<td>1.5714e-04</td>
</tr>
<tr>
<td>0.60</td>
<td>0.002713</td>
<td>0.002680</td>
<td>3.2810e-05</td>
</tr>
<tr>
<td>0.80</td>
<td>0.000378</td>
<td>0.000372</td>
<td>6.0893e-06</td>
</tr>
<tr>
<td>1.00</td>
<td>0.000059</td>
<td>0.000052</td>
<td>1.0595e-06</td>
</tr>
</tbody>
</table>

### rkc4b integrator

**ncase = 4  neqn = 21  nsteps = 250**

<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.00000000</td>
<td>1.00000000</td>
<td>0.0000e+00</td>
</tr>
<tr>
<td>0.20</td>
<td>0.139479</td>
<td>0.138911</td>
<td>5.6748e-04</td>
</tr>
<tr>
<td>0.40</td>
<td>0.019454</td>
<td>0.019296</td>
<td>1.5742e-04</td>
</tr>
<tr>
<td>0.60</td>
<td>0.002714</td>
<td>0.002680</td>
<td>3.3680e-05</td>
</tr>
<tr>
<td>0.80</td>
<td>0.000380</td>
<td>0.000372</td>
<td>7.3932e-06</td>
</tr>
<tr>
<td>1.00</td>
<td>0.000054</td>
<td>0.000052</td>
<td>2.0762e-06</td>
</tr>
</tbody>
</table>
rkf45a integrator

ncase = 5  neqn = 21  nsteps = 250

<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.0000e+00</td>
</tr>
<tr>
<td>0.20</td>
<td>0.139476</td>
<td>0.138911</td>
<td>5.6448e-04</td>
</tr>
<tr>
<td>0.40</td>
<td>0.019453</td>
<td>0.019296</td>
<td>1.5714e-04</td>
</tr>
<tr>
<td>0.60</td>
<td>0.002713</td>
<td>0.002680</td>
<td>3.2810e-05</td>
</tr>
<tr>
<td>0.80</td>
<td>0.000378</td>
<td>0.000372</td>
<td>6.0893e-06</td>
</tr>
<tr>
<td>1.00</td>
<td>0.000053</td>
<td>0.000052</td>
<td>1.0595e-06</td>
</tr>
</tbody>
</table>

rkf45b integrator

ncase = 6  neqn = 21  nsteps = 250

<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.0000e+00</td>
</tr>
<tr>
<td>0.20</td>
<td>0.139484</td>
<td>0.138911</td>
<td>5.7324e-04</td>
</tr>
<tr>
<td>0.40</td>
<td>0.019453</td>
<td>0.019296</td>
<td>1.5794e-04</td>
</tr>
<tr>
<td>0.60</td>
<td>0.002713</td>
<td>0.002680</td>
<td>3.2809e-05</td>
</tr>
<tr>
<td>0.80</td>
<td>0.000378</td>
<td>0.000372</td>
<td>6.0901e-06</td>
</tr>
<tr>
<td>1.00</td>
<td>0.000051</td>
<td>0.000052</td>
<td>-5.0979e-07</td>
</tr>
</tbody>
</table>

However, we note that the numerical solution does not meet the error criteria set in intpar

% Error tolerances
  abserr=1.0e-05;
  relerr=1.0e-05;

For example, from euler2a, the following output has errors (diff) that exceed the error tolerances:

<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.0000e+00</td>
</tr>
<tr>
<td>0.20</td>
<td>0.139478</td>
<td>0.138911</td>
<td>5.6734e-04</td>
</tr>
<tr>
<td>0.40</td>
<td>0.019454</td>
<td>0.019296</td>
<td>1.5794e-04</td>
</tr>
<tr>
<td>0.60</td>
<td>0.002713</td>
<td>0.002680</td>
<td>3.2977e-05</td>
</tr>
<tr>
<td>0.80</td>
<td>0.000378</td>
<td>0.000372</td>
<td>6.0901e-06</td>
</tr>
<tr>
<td>1.00</td>
<td>0.000051</td>
<td>0.000052</td>
<td>-5.0979e-07</td>
</tr>
</tbody>
</table>

However, this does not mean that euler2a did not integrate correctly because now we have a second source of error in the numerical solution, the spatial discretization error from the FD approximation of Equation 4.8.
To investigate this possible explanation for the errors in the output, the preceding MATLAB program was executed with `neqn = 31` in `intpar` (31 grid points). Again the solution became unstable, and it was necessary to increase `nsteps` to 500. When this was done, the output listed below resulted:

```
euler2a integrator
ncase = 1  neqn = 31  nsteps = 500
```

<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.0000e+00</td>
</tr>
<tr>
<td>0.20</td>
<td>0.139163</td>
<td>0.138911</td>
<td>2.5143e-04</td>
</tr>
<tr>
<td>0.40</td>
<td>0.019366</td>
<td>0.019296</td>
<td>6.9915e-05</td>
</tr>
<tr>
<td>0.60</td>
<td>0.002695</td>
<td>0.002680</td>
<td>1.4581e-05</td>
</tr>
<tr>
<td>0.80</td>
<td>0.000375</td>
<td>0.000372</td>
<td>2.7031e-06</td>
</tr>
<tr>
<td>1.00</td>
<td>0.000052</td>
<td>0.000052</td>
<td>4.6979e-07</td>
</tr>
</tbody>
</table>

```
euler2b integrator
ncase = 2  neqn = 31  nsteps = 500
```

<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.0000e+00</td>
</tr>
<tr>
<td>0.20</td>
<td>0.139182</td>
<td>0.138911</td>
<td>2.7122e-04</td>
</tr>
<tr>
<td>0.40</td>
<td>0.019369</td>
<td>0.019296</td>
<td>7.3151e-05</td>
</tr>
<tr>
<td>0.60</td>
<td>0.002695</td>
<td>0.002680</td>
<td>1.4604e-05</td>
</tr>
<tr>
<td>0.80</td>
<td>0.000365</td>
<td>0.000372</td>
<td>-6.9759e-06</td>
</tr>
<tr>
<td>1.00</td>
<td>0.000042</td>
<td>0.000052</td>
<td>-1.0031e-05</td>
</tr>
</tbody>
</table>

```
rkc4a integrator
ncase = 3  neqn = 31  nsteps = 500
```

<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.0000e+00</td>
</tr>
<tr>
<td>0.20</td>
<td>0.139162</td>
<td>0.138911</td>
<td>2.5071e-04</td>
</tr>
<tr>
<td>0.40</td>
<td>0.019366</td>
<td>0.019296</td>
<td>6.9716e-05</td>
</tr>
<tr>
<td>0.60</td>
<td>0.002695</td>
<td>0.002680</td>
<td>1.4540e-05</td>
</tr>
<tr>
<td>0.80</td>
<td>0.000375</td>
<td>0.000372</td>
<td>2.6954e-06</td>
</tr>
<tr>
<td>1.00</td>
<td>0.000052</td>
<td>0.000052</td>
<td>4.6845e-07</td>
</tr>
</tbody>
</table>

rkc4b integrator

case = 4  neqn = 31  nsteps = 500

<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.0000e+00</td>
</tr>
<tr>
<td>0.20</td>
<td>0.139159</td>
<td>0.138911</td>
<td>2.4782e-04</td>
</tr>
<tr>
<td>0.40</td>
<td>0.019221</td>
<td>0.019296</td>
<td>-7.5061e-05</td>
</tr>
<tr>
<td>0.60</td>
<td>0.002698</td>
<td>0.002680</td>
<td>1.7490e-05</td>
</tr>
<tr>
<td>0.80</td>
<td>0.000327</td>
<td>0.000372</td>
<td>-4.4904e-05</td>
</tr>
<tr>
<td>1.00</td>
<td>0.000042</td>
<td>0.000052</td>
<td>-9.3554e-06</td>
</tr>
</tbody>
</table>

rkf45a integrator

case = 5  neqn = 31  nsteps = 500

<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.0000e+00</td>
</tr>
<tr>
<td>0.20</td>
<td>0.139162</td>
<td>0.138911</td>
<td>2.5071e-04</td>
</tr>
<tr>
<td>0.40</td>
<td>0.019366</td>
<td>0.019296</td>
<td>6.9716e-05</td>
</tr>
<tr>
<td>0.60</td>
<td>0.002695</td>
<td>0.002680</td>
<td>1.4540e-05</td>
</tr>
<tr>
<td>0.80</td>
<td>0.000375</td>
<td>0.000372</td>
<td>2.6954e-06</td>
</tr>
<tr>
<td>1.00</td>
<td>0.000052</td>
<td>0.000052</td>
<td>4.6845e-07</td>
</tr>
</tbody>
</table>

rkf45b integrator

case = 6  neqn = 31  nsteps = 500

<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.0000e+00</td>
</tr>
<tr>
<td>0.20</td>
<td>0.139161</td>
<td>0.138911</td>
<td>2.5071e-04</td>
</tr>
<tr>
<td>0.40</td>
<td>0.019368</td>
<td>0.019296</td>
<td>6.9716e-05</td>
</tr>
<tr>
<td>0.60</td>
<td>0.002690</td>
<td>0.002680</td>
<td>9.5294e-06</td>
</tr>
<tr>
<td>0.80</td>
<td>0.000389</td>
<td>0.000372</td>
<td>1.6293e-05</td>
</tr>
<tr>
<td>1.00</td>
<td>0.000052</td>
<td>0.000052</td>
<td>6.4318e-08</td>
</tr>
</tbody>
</table>

Note now that the errors are smaller, e.g., for euler2a;

neqn = 21

<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.0000e+00</td>
</tr>
<tr>
<td>0.20</td>
<td>0.139478</td>
<td>0.138911</td>
<td>5.6734e-04</td>
</tr>
</tbody>
</table>
Since the FD approximation of Equation 4.8 is $O(\Delta x^2)$, we can estimate the error $\epsilon_{31}$ for $neqn = 31$ from the error $\epsilon_{21}$ for $neqn = 21$:

\[\epsilon_{31} = c_{31}\Delta x^2_{31}\]
\[\epsilon_{21} = c_{21}\Delta x^2_{21}\]

Division of one equation by the other, assuming $c_{21} = c_{31}$ gives

\[\frac{\epsilon_{31}}{\epsilon_{21}} = \frac{\Delta x^2_{31}}{\Delta x^2_{21}}\]

and therefore

\[\epsilon_{31} = \epsilon_{21}\left[\frac{\Delta x_{31}}{\Delta x_{21}}\right]^2\]

We can apply this result to estimate $\epsilon_{31}$. For $neqn = 21$, $\Delta x_{21} = (1 - 0)/(21 - 1)$ and for $neqn = 31$, $\Delta x_{31} = (1 - 0)/(31 - 1)$. Therefore, $\Delta x_{31}/\Delta x_{21} = (21 - 1)/(31 - 1)$. For $t = 0.20$ (using numbers from the preceding output)

\[5.6734e - 04 \times [(21 - 1)/(31 - 1)]^2 = 2.5219e - 04 \simeq 2.5143e - 04\]

Thus, we can infer that the errors above the error tolerances set in intpar are due to the spatial gridding or discretization, i.e., the choice of a low value of $neqn$. In other words, if the integration $t$ contributed a significant error to the preceding output, the $O(\Delta x^2)$ analysis would not necessarily give these consistent numerical results.

This conclusion was explored further by computing a solution at $neqn = 41$ (and $nsteps = 500$) in intpar. Again the solution became unstable, but by using $nsteps = 1000$, the solution stabilized to the values indicated below:

<table>
<thead>
<tr>
<th>$t$</th>
<th>$u$ (num)</th>
<th>$u$ (exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.0000e+00</td>
</tr>
<tr>
<td>0.20</td>
<td>0.139163</td>
<td>0.138911</td>
<td>2.5143e-04</td>
</tr>
<tr>
<td>0.40</td>
<td>0.019366</td>
<td>0.019296</td>
<td>6.9915e-05</td>
</tr>
<tr>
<td>0.60</td>
<td>0.002695</td>
<td>0.002680</td>
<td>1.4581e-05</td>
</tr>
</tbody>
</table>

Copyright © 2004 by Chapman & Hall/CRC
### euler2a integrator

<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.000000e+00</td>
</tr>
<tr>
<td>0.20</td>
<td>0.139052</td>
<td>0.138911</td>
<td>1.4117e-04</td>
</tr>
<tr>
<td>0.40</td>
<td>0.019336</td>
<td>0.019296</td>
<td>3.9240e-05</td>
</tr>
<tr>
<td>0.60</td>
<td>0.002689</td>
<td>0.002680</td>
<td>8.1805e-06</td>
</tr>
<tr>
<td>0.80</td>
<td>0.000374</td>
<td>0.000372</td>
<td>1.5159e-06</td>
</tr>
<tr>
<td>1.00</td>
<td>0.000052</td>
<td>0.000052</td>
<td>2.6336e-07</td>
</tr>
</tbody>
</table>

### euler2b integrator

<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.000000e+00</td>
</tr>
<tr>
<td>0.20</td>
<td>0.139235</td>
<td>0.138911</td>
<td>3.2417e-04</td>
</tr>
<tr>
<td>0.40</td>
<td>0.019400</td>
<td>0.019296</td>
<td>1.0374e-04</td>
</tr>
<tr>
<td>0.60</td>
<td>0.002701</td>
<td>0.002680</td>
<td>2.0959e-05</td>
</tr>
<tr>
<td>0.80</td>
<td>0.000374</td>
<td>0.000372</td>
<td>1.9047e-06</td>
</tr>
<tr>
<td>1.00</td>
<td>0.000046</td>
<td>0.000052</td>
<td>-5.8058e-06</td>
</tr>
</tbody>
</table>

### rkc4a integrator

<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.000000e+00</td>
</tr>
<tr>
<td>0.20</td>
<td>0.139052</td>
<td>0.138911</td>
<td>3.9191e-05</td>
</tr>
<tr>
<td>0.40</td>
<td>0.019335</td>
<td>0.019296</td>
<td>1.4099e-04</td>
</tr>
<tr>
<td>0.60</td>
<td>0.002689</td>
<td>0.002680</td>
<td>8.1702e-06</td>
</tr>
<tr>
<td>0.80</td>
<td>0.000374</td>
<td>0.000372</td>
<td>1.5140e-06</td>
</tr>
<tr>
<td>1.00</td>
<td>0.000052</td>
<td>0.000052</td>
<td>2.6302e-07</td>
</tr>
</tbody>
</table>

### rkc4b integrator

<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.000000e+00</td>
</tr>
<tr>
<td>0.20</td>
<td>0.139118</td>
<td>0.138911</td>
<td>2.0675e-04</td>
</tr>
</tbody>
</table>
### rkf45a integrator

```latex
case = 5 \quad neqn = 41 \quad nsteps = 1000
```

<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.0000e+00</td>
</tr>
<tr>
<td>0.20</td>
<td>0.139052</td>
<td>0.138911</td>
<td>1.4099e-04</td>
</tr>
<tr>
<td>0.40</td>
<td>0.019335</td>
<td>0.019296</td>
<td>3.9191e-05</td>
</tr>
<tr>
<td>0.60</td>
<td>0.002689</td>
<td>0.002680</td>
<td>8.1702e-06</td>
</tr>
<tr>
<td>0.80</td>
<td>0.000374</td>
<td>0.000372</td>
<td>1.5140e-06</td>
</tr>
<tr>
<td>1.00</td>
<td>0.000052</td>
<td>0.000052</td>
<td>2.6302e-07</td>
</tr>
</tbody>
</table>

### rkf45b integrator

```latex
case = 6 \quad neqn = 41 \quad nsteps = 1000
```

<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.0000e+00</td>
</tr>
<tr>
<td>0.20</td>
<td>0.139052</td>
<td>0.138911</td>
<td>1.4099e-04</td>
</tr>
<tr>
<td>0.40</td>
<td>0.019340</td>
<td>0.019296</td>
<td>4.4135e-05</td>
</tr>
<tr>
<td>0.60</td>
<td>0.002694</td>
<td>0.002680</td>
<td>1.3738e-05</td>
</tr>
<tr>
<td>0.80</td>
<td>0.000381</td>
<td>0.000372</td>
<td>9.0848e-06</td>
</tr>
<tr>
<td>1.00</td>
<td>0.000048</td>
<td>0.000052</td>
<td>-3.6816e-06</td>
</tr>
</tbody>
</table>

Again, the errors are reduced below those for \( neqn = 21 \) and \( neqn = 31 \):

### neqn = 21

```latex
```

<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.0000e+00</td>
</tr>
<tr>
<td>0.20</td>
<td>0.139478</td>
<td>0.138911</td>
<td>5.6734e-04</td>
</tr>
<tr>
<td>0.40</td>
<td>0.019454</td>
<td>0.019296</td>
<td>1.5794e-04</td>
</tr>
<tr>
<td>0.60</td>
<td>0.002713</td>
<td>0.002680</td>
<td>3.2977e-05</td>
</tr>
</tbody>
</table>

### neqn = 31

```latex
```

<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.0000e+00</td>
</tr>
<tr>
<td>0.20</td>
<td>0.139163</td>
<td>0.138911</td>
<td>2.5143e-04</td>
</tr>
</tbody>
</table>
neqn=41

<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.0000e+00</td>
</tr>
<tr>
<td>0.20</td>
<td>0.139052</td>
<td>0.138911</td>
<td>1.4117e-04</td>
</tr>
<tr>
<td>0.40</td>
<td>0.019336</td>
<td>0.019296</td>
<td>3.9240e-05</td>
</tr>
<tr>
<td>0.60</td>
<td>0.002689</td>
<td>0.002680</td>
<td>8.1805e-06</td>
</tr>
</tbody>
</table>

The error analysis at \( t = 0.20 \) gives

\[
5.6734e-04 \times [(21 - 1)/(41 - 1)]^2 = 1.4184e-04 \simeq 1.4117e-04
\]

Again, this error analysis using \( O(\Delta x^2) \) is consistent with the dominant effect of the truncation errors in \( x \) (the error in \( t \) is relatively small).

To reiterate the principal points of this analysis:

- By increasing \( \text{neqn} \) (the number of grid points) and thereby decreasing the spatial discretization error in \( x \), we would eventually reach a point where the truncation error in \( t \) might become comparable to the error in \( x \) (assuming we could keep the solution stable by increasing \( \text{nsteps} \)). In other words, the errors in the \( x \) and \( t \) integrations occur simultaneously.

- To restate the preceding point, we were able to perform the error analysis in this case (using \( O(\Delta x^2) \) from Equation 4.9) only because the spatial error in \( x \) dominated the solution error. We in effect varied \( \Delta x \) in Equation 4.9 (by changing \( \text{neqn} \)) to estimate the spatial error while the error due to \( h \) in the \( t \) integration remained relatively small (or at least smaller than the \( \Delta x \) error in the \( x \) integration).

- As the grid is refined in \( x \) (by increasing \( \text{neqn} \)), the resulting system of ODEs becomes \textit{larger and stiffer} (the increased stiffness was apparent from the need to increase \( \text{nsteps} \) or reduce \( h \)). The separation in the ODE eigenvalues is proportional to \( 1/\Delta x^2 \), and as \( \Delta x \) becomes smaller (with increasing \( \text{neqn} \)), the \textit{ODE eigenvalues become more widely separated} (and so the ODE system becomes stiffer).

- Thus, there is a twofold effect that increases the computational effort, \textit{increased number of ODEs and increase in stiffness of these ODEs}. This increased computational effort to improve the accuracy in \( x \) is apparent in running the preceding program with increased \( \text{neqn} \); i.e., it runs perceptibly slower.

As a second interesting observation (in addition to instability), we observed that the lower-order algorithms (in \textit{euler2a} and \textit{euler2b}) became unstable while the higher-order algorithms (in \textit{rkc4a}, \textit{rkc4b}, \textit{rkf45a}, and \textit{rkf45b}) generally remained stable. This effect is due to the slightly larger stability domain for the
higher-order methods, e.g., $|\lambda h| < 2$ for the Euler method and $|\lambda h| < 2.785$ for the classical fourth-order RK method as determined from Equation 55c.

As a third observation, the variable step integrators, euler2b, rkc4b, and rkf45b executed perceptibly faster than the fixed step integrators, euler2a, rkc4a, and rkf45a. This can be explained by the use of the smallest integration step, $h = (t_f - t_0)/nsteps$, throughout the solution by the fixed step integrators, while the variable step integrators could increase $h$ above this value, and thus, by taking larger steps, the solutions could be computed faster. In other words, this observation clearly demonstrates the relative computational efficiency of the variable step integrators.

To summarize the preceding observations, when numerically integrating PDEs, we must investigate and control the errors in time and space (e.g., $t$ and $x$) to arrive at solutions with acceptable accuracy. This investigation usually takes the form of numerical experimentation, since in most cases, we will not have analytical solutions (like Equation 4.7) to give us the exact errors; i.e., we can only estimate the errors in time and space. More advanced integration algorithms in $x$ and $t$ could, at least in principle, adjust $h$ and $\Delta x$ automatically to achieve solutions of prescribed accuracy, but we should still keep in mind the requirement to control truncation errors in time and space, and to carefully examine and test the solutions.

This completes the solution of Equations 4.1 to 4.4 with MATLAB. We now proceed to equivalent solutions in the five other languages.

### 4.2 Programming in C

Since main Program 2.2.1 is unchanged in the nonlinear PDE problem, it is not listed here. The header file, intpar, par, initial, derv, and fprint are listed below:

```c
/*

Definition of functions intpar, par, initial, derv, fprint
for the linear PDE

*/

/* Include headers */
#include <math.h>
#include <stdio.h>

/* Maximum number of ODES */
#define SIZE 500
```
/* Type variables as extern (global) */
extern int neqn, nout, nsteps;
extern double t0, tf, abserr, relerr;
extern double pi;
extern double xl, xu, dx, dxs;
extern FILE *fid;

/* Integration parameters */
void intpar();

/* Problem parameters */
void par();

/* Initial condition */
void initial(double u0[]);

/* Derivative vector */
void deriv(double ut[], double t, double u[]);

/* Output */
void fprint(int ncase, double t, double u[]);

#include "pdelin.h"

#define SIZE 500

/* Type global variables */
int neqn, nout, nsteps;
double t0, tf, abserr, relerr;
double pi, xl, xu, dx, dxs;

/* Define file ID */
FILE *fid;

void intpar()

/* Function intpar sets the parameters to control
the integration of the linear PDE system */
{
/ * Number of ODEs */
neqn=21;

/* Number of output points */
	nout=6;

/* Maximum number of steps in the interval t0 to tf */

nsteps=250;

/* Initial, final values of the independent variable */
t0=0.0;

tf=0.2;

/* Error tolerances */

abserr=pow(10.0,-5);

relerr=pow(10.0,-5);

/* End of intpar */

}

void par()

/* Function par sets the parameters for the linear PDE problem */

{

pi=4.0*atan(1.0);

xl=0.0;

xu=1.0;

dx=(xu-xl)/(neqn-1);

dxs=dx*dx;

}

void inital(double u0[])

/* Function initial sets the initial condition vector for the linear PDE problem */

{

/* Type variables */

int i;

double x;

/* Problem parameters */

par();
/* Initial condition */
for(i=1;i<=neqn;i++)
{
    x=xl+(float)(i-1)/(neqn-1)*(xu-xl);
    u0[i]=sin(pi*x);
}
/* End of initial */

void derv(double ut[], double t, double u[])

/* Function derv computes the derivative vector of the linear PDE problem */
{
    /* Type variables */
    int i;

    /* Problem parameters */
    par();

    /* BC at x = xl */
    ut[1]=0.0;

    /* BC at x = xu */
    ut[neqn]=0.0;

    /* Derivative vector */
    for(i=2;i<=(neqn-1);i++)
    {
        /* Approximation to the PDE */
        ut[i]=(u[i+1]-2.0*u[i]+u[i-1])/dxs;
    }
    /* End of derv */
}

void fprint(int ncase, double t, double u[])

/* Function fprint displays the numerical and exact solutions to the linear PDE problem */
{
    /* Type variables */
    double x, ue;
    int im;
/* Print a heading for the solution at t = 0 */
if(t<=0.0)
{
    /* Label for ODE integrator */
    switch(ncase)
    {
        /*Fixed step modified Euler */
        case 1:
            fprintf(fid,"\n\n euler2a integrator\n\n");
            break;
        /* Variable step modified Euler */
        case 2:
            fprintf(fid,"\n\n euler2b integrator\n\n");
            break;
        /* Fixed step classical fourth order RK */
        case 3:
            fprintf(fid,"\n\n rkc4a integrator\n\n");
            break;
        /* Variable step classical fourth order RK */
        case 4:
            fprintf(fid,"\n\n rkc4b integrator\n\n");
            break;
        /* Fixed step RK Fehlberg 45 */
        case 5:
            fprintf(fid,"\n\n rkf45a integrator\n\n");
            break;
        /* Variable step RK Fehlberg 45 */
        case 6:
            fprintf(fid,"\n\n rkf45b integrator\n\n");
            break;
    }
    /* Heading */
    fprintf(fid," ncase = %1d neqn = %3d\n",ncase,neqn);
    fprintf(fid," t   u(0.5)   ue(0.5)   diff\n");

    /* End of t = 0 heading */
}
/ * Numerical and analytical solution output */

/* Problem parameters */
par();

/* Midpoint value of x */
x=(xu-xl)/2.0;

/* Analytical solution at midpoint */
ue=exp(-pi*pi*t)*sin(pi*x);

/* Grid index of midpoint */
im=(neqn+1)/2;

/* Display the numerical and exact solutions, and their difference */
fprintf(fid,"%5.2f %11.6f %11.6f %13.4e\n",t,u[im],ue,u[im]-ue);

/* End of fprintf */
}

Program 4.2.1
Header file, intpar, par, initial, derv, and fprintf for the solution of Equations 4.1 to 4.4

The preceding intpar, par, initial, derv, and fprintf closely parallel the analogous routines of Programs 4.1.1 to 4.1.4; therefore, no further discussion is given here, except to point out that a routine par was used to set the problem parameters in Program 4.2.1 while the parameters were set in each routine where they are used in Programs 4.1.2 to 4.1.4.

The output from the routines of Program 4.2.1 is listed below:
euler2a integrator
ncase = 1 neqn = 21

<table>
<thead>
<tr>
<th>t</th>
<th>u(0.5)</th>
<th>ue(0.5)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.0000e+00</td>
</tr>
<tr>
<td>0.20</td>
<td>0.139478</td>
<td>0.138911</td>
<td>5.6734e-04</td>
</tr>
<tr>
<td>0.40</td>
<td>0.019454</td>
<td>0.019296</td>
<td>1.5794e-04</td>
</tr>
<tr>
<td>0.60</td>
<td>0.002713</td>
<td>0.002680</td>
<td>2.977e-05</td>
</tr>
<tr>
<td>0.80</td>
<td>0.000378</td>
<td>0.000372</td>
<td>6.1203e-06</td>
</tr>
<tr>
<td>1.00</td>
<td>0.000053</td>
<td>0.000052</td>
<td>1.0649e-06</td>
</tr>
</tbody>
</table>

Copyright 2004 by Chapman & Hall/CRC
### euler2b integrator

\( ncase = 2 \quad neqn = 21 \)

<table>
<thead>
<tr>
<th>t</th>
<th>u(0.5)</th>
<th>ue(0.5)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.0000e+00</td>
</tr>
<tr>
<td>0.20</td>
<td>0.139530</td>
<td>0.138911</td>
<td>6.1894e-04</td>
</tr>
<tr>
<td>0.40</td>
<td>0.019466</td>
<td>0.019296</td>
<td>1.7004e-04</td>
</tr>
<tr>
<td>0.60</td>
<td>0.002713</td>
<td>0.002680</td>
<td>3.2297e-05</td>
</tr>
<tr>
<td>0.80</td>
<td>0.000378</td>
<td>0.000372</td>
<td>5.7139e-06</td>
</tr>
<tr>
<td>1.00</td>
<td>0.000047</td>
<td>0.000052</td>
<td>-4.3060e-06</td>
</tr>
</tbody>
</table>

### rkc4a integrator

\( ncase = 3 \quad neqn = 21 \)

<table>
<thead>
<tr>
<th>t</th>
<th>u(0.5)</th>
<th>ue(0.5)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.0000e+00</td>
</tr>
<tr>
<td>0.20</td>
<td>0.139476</td>
<td>0.138911</td>
<td>5.6448e-04</td>
</tr>
<tr>
<td>0.40</td>
<td>0.019453</td>
<td>0.019296</td>
<td>1.5714e-04</td>
</tr>
<tr>
<td>0.60</td>
<td>0.002713</td>
<td>0.002680</td>
<td>3.2810e-05</td>
</tr>
<tr>
<td>0.80</td>
<td>0.000378</td>
<td>0.000372</td>
<td>6.0893e-06</td>
</tr>
<tr>
<td>1.00</td>
<td>0.000053</td>
<td>0.000052</td>
<td>1.0595e-06</td>
</tr>
</tbody>
</table>

### rkc4b integrator

\( ncase = 4 \quad neqn = 21 \)

<table>
<thead>
<tr>
<th>t</th>
<th>u(0.5)</th>
<th>ue(0.5)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.0000e+00</td>
</tr>
<tr>
<td>0.20</td>
<td>0.139476</td>
<td>0.138911</td>
<td>5.6480e-04</td>
</tr>
<tr>
<td>0.40</td>
<td>0.019454</td>
<td>0.019296</td>
<td>1.5815e-04</td>
</tr>
<tr>
<td>0.60</td>
<td>0.002717</td>
<td>0.002680</td>
<td>3.6838e-05</td>
</tr>
<tr>
<td>0.80</td>
<td>0.000380</td>
<td>0.000372</td>
<td>7.9663e-06</td>
</tr>
<tr>
<td>1.00</td>
<td>0.000054</td>
<td>0.000052</td>
<td>2.2905e-06</td>
</tr>
</tbody>
</table>

### rkf45a integrator

\( ncase = 5 \quad neqn = 21 \)

<table>
<thead>
<tr>
<th>t</th>
<th>u(0.5)</th>
<th>ue(0.5)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.0000e+00</td>
</tr>
<tr>
<td>0.20</td>
<td>0.139476</td>
<td>0.138911</td>
<td>5.6448e-04</td>
</tr>
</tbody>
</table>
This output is essentially the same as the MATLAB output in Section 4.1, i.e., the error tolerances in intpar are exceeded because of the spatial discretization errors. We now proceed to the C++ programming of the linear PDE.

### 4.3 Programming in C++

Again, since main Program 2.3.1 is unchanged, it is not listed here. The associated header files are listed below:

```
/*
    Definition of functions intpar, initial, par, derv, fprint
    for the linear PDE system

    */
```

```c
#include <stdio.h>
#include <math.h>
#include "MOL.h"
/* Type variables as extern (global) */
extern FILE *fid;
```
class DEF:public MOL
{
    public:

/* Integration parameters */
    void intpar();

/* Problem parameters */
    void par();

/* Initial condition */
    void initial();

/* Derivative vector */
    void derv(double ut[], double t, double u[]);

/* Output */
    void fprint(int ncase, int neqn, double t, double u[]);
};

/* Define the common (global) variables for the linear PDE integration */

/* Maximum (default) number of ODEs */
#define SIZE 500

class MOL
{
    public:

/* Variables for ODE integration */
    int neqn, nout, nsteps;
    double t0, tf, abserr, relerr;
    double u[SIZE], u0[SIZE], e[SIZE];
/* Problem parameters */

double pi, xl, xu, dx, dxs;

}

**Program 4.3.1**

Header files *pdelin.h* and *MOL.h* for the solution of Equations 4.1 to 4.4

`intpar, par, inital, derv, and fprint` are listed below:

```c
#include "DEF.h"

void DEF::intpar()

/* Function intpar sets the parameters to control the
integration of the linear PDE system */
{
    /* Number of ODEs */
    neqn=21;

    /* Number of output points */
    nout=6;

    /* Maximum number of steps in the interval t0 to tf */
    nsteps=250;

    /* Initial, final values of the independent variable */
    t0=0.0;
    tf=0.2;

    /* Error tolerances */
    abserr=pow(10.0,-5);
    relerr=pow(10.0,-5);

    /* End of intpar */
}

void DEF::par()

/* Function par sets the parameters for the linear PDE
problem */
{
    pi=4.0*atan(1.0);
    xl=0.0;
```
void DEF::inital()

/* Function inital sets the initial condition vector for
the linear PDE problem */
{
    /* Type variables */
    double x;
    int i;

    /* Initial condition */
    for(i=1;i<=neqn;i++)
    {
        x=xl+(float)(i-1)/(neqn-1)*(xu-xl);
        u0[i]=sin(pi*x);
    }

    /* End of inital */
}

void DEF::derv(double ut[], double t, double u[])

/* Function derv computes the derivative vector of the
linear PDE problem */
{
    /* Type variables */
    int i;

    /* Integration parameters */
    intpar();

    /* Problem parameters */
    par();

    /* BC at x = xl */
    ut[1]=0.0;

    /* BC at x = xu */
    ut[neqn]=0.0;
/* Derivative vector */
for(i=2;i<=(neqn-1);i++)
{
    /* Approximation to the PDE */
    ut[i]=(u[i+1]-2.0*u[i]+u[i-1])/dxs;
}
/* End of derv */

void DEF::fprint(int ncase, int neqn, double t, double u[])
/* Function fprint displays the numerical and exact solutions to the linear PDE problem */
{
    /* Type variables */
    double x, ue;
    int im;
    /* Problem parameters */
    par();
    /* Print a heading for the solution at t = 0 */
    if(t<=0.0)
    {
        /* Label for ODE integrator */
        switch(ncase)
        {
            /*Fixed step modified Euler */
            case 1:
                fprintf(fid,"\n\n euler2a integrator\n\n");
                break;
            /* Variable step modified Euler */
            case 2:
                fprintf(fid,"\n\n euler2b integrator\n\n");
                break;
            /* Fixed step classical fourth order RK */
            case 3:
                fprintf(fid,"\n\n rkc4a integrator\n\n");
                break;
        }
    }
/* Variable step classical fourth order RK */
case 4:
    fprintf(fid, "\n\n rkc4b integrator\n\n");
    break;

/* Fixed step RK Fehlberg 45 */
case 5:
    fprintf(fid, "\n\n rkf45a integrator\n\n");
    break;

/* Variable step RK Fehlberg 45 */
case 6:
    fprintf(fid, "\n\n rkf45b integrator\n\n");
    break;
}

/* Heading */
    fprintf(fid,"ncase = %ld, neqn = %ld \n\n",ncase,n);
fprintf(fid,"t u(num) u(exact) diff\n");

/* End of t = 0 heading */
}

/* Numerical and analytical solution output */

/* Midpoint value of x */
    x=(xu-xl)/2.0;

/* Analytical solution at midpoint */
    ue=exp(-pi*pi*t)*sin(pi*x);

/* Grid index of midpoint */
    im=(neqn+1)/2;

/* Display the numerical and exact solutions,
and their difference */
    fprintf(fid,"%5.2f %11.6f %11.6f %13.4e\n",t,u[im],ue,u[im]-ue);

/* End of fprintf */
}

Program 4.3.2
intpar, par, initial, derv, and fprint for the solution of Equations 4.1 to 4.4
The output from the routines of Programs 4.3.1 and 4.3.2 is listed below:

euler2a integrator
ncase = 1, neqn = 21

<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.0000e+00</td>
</tr>
<tr>
<td>0.20</td>
<td>0.139478</td>
<td>0.138911</td>
<td>5.6734e-04</td>
</tr>
<tr>
<td>0.40</td>
<td>0.019454</td>
<td>0.019296</td>
<td>1.5794e-04</td>
</tr>
<tr>
<td>0.60</td>
<td>0.002713</td>
<td>0.002680</td>
<td>3.2977e-05</td>
</tr>
<tr>
<td>0.80</td>
<td>0.000378</td>
<td>0.000372</td>
<td>6.1203e-06</td>
</tr>
<tr>
<td>1.00</td>
<td>0.000053</td>
<td>0.000052</td>
<td>1.0649e-06</td>
</tr>
</tbody>
</table>

euler2b integrator
ncase = 2, neqn = 21

<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.0000e+00</td>
</tr>
<tr>
<td>0.20</td>
<td>0.139530</td>
<td>0.138911</td>
<td>6.1894e-04</td>
</tr>
<tr>
<td>0.40</td>
<td>0.019466</td>
<td>0.019296</td>
<td>1.7004e-04</td>
</tr>
<tr>
<td>0.60</td>
<td>0.002713</td>
<td>0.002680</td>
<td>3.2297e-05</td>
</tr>
<tr>
<td>0.80</td>
<td>0.000378</td>
<td>0.000372</td>
<td>5.7139e-06</td>
</tr>
<tr>
<td>1.00</td>
<td>0.000047</td>
<td>0.000052</td>
<td>-4.3060e-06</td>
</tr>
</tbody>
</table>

rkc4a integrator
ncase = 3, neqn = 21

<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.0000e+00</td>
</tr>
<tr>
<td>0.20</td>
<td>0.139476</td>
<td>0.138911</td>
<td>5.6448e-04</td>
</tr>
<tr>
<td>0.40</td>
<td>0.019453</td>
<td>0.019296</td>
<td>1.5714e-04</td>
</tr>
<tr>
<td>0.60</td>
<td>0.002713</td>
<td>0.002680</td>
<td>3.2810e-05</td>
</tr>
<tr>
<td>0.80</td>
<td>0.000378</td>
<td>0.000372</td>
<td>6.0893e-06</td>
</tr>
<tr>
<td>1.00</td>
<td>0.000053</td>
<td>0.000052</td>
<td>1.0595e-06</td>
</tr>
</tbody>
</table>
rkc4b integrator
ncase = 4, neqn = 21

<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.00000e+00</td>
</tr>
<tr>
<td>0.20</td>
<td>0.139476</td>
<td>0.138911</td>
<td>5.6480e-04</td>
</tr>
<tr>
<td>0.40</td>
<td>0.019454</td>
<td>0.019296</td>
<td>1.5815e-04</td>
</tr>
<tr>
<td>0.60</td>
<td>0.002717</td>
<td>0.002680</td>
<td>3.6838e-05</td>
</tr>
<tr>
<td>0.80</td>
<td>0.000380</td>
<td>0.000372</td>
<td>7.9663e-06</td>
</tr>
<tr>
<td>1.00</td>
<td>0.000054</td>
<td>0.000052</td>
<td>2.2905e-06</td>
</tr>
</tbody>
</table>

rkf45a integrator
ncase = 5, neqn = 21

<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.00000e+00</td>
</tr>
<tr>
<td>0.20</td>
<td>0.139476</td>
<td>0.138911</td>
<td>5.6448e-04</td>
</tr>
<tr>
<td>0.40</td>
<td>0.019453</td>
<td>0.019296</td>
<td>1.5714e-04</td>
</tr>
<tr>
<td>0.60</td>
<td>0.002713</td>
<td>0.002680</td>
<td>3.2810e-05</td>
</tr>
<tr>
<td>0.80</td>
<td>0.000378</td>
<td>0.000372</td>
<td>6.0893e-06</td>
</tr>
<tr>
<td>1.00</td>
<td>0.000053</td>
<td>0.000052</td>
<td>1.0595e-06</td>
</tr>
</tbody>
</table>

rKF45b integrator
ncase = 6, neqn = 21

<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.00000e+00</td>
</tr>
<tr>
<td>0.20</td>
<td>0.139476</td>
<td>0.138911</td>
<td>5.6472e-04</td>
</tr>
<tr>
<td>0.40</td>
<td>0.019454</td>
<td>0.019296</td>
<td>1.5721e-04</td>
</tr>
<tr>
<td>0.60</td>
<td>0.002713</td>
<td>0.002680</td>
<td>3.3024e-05</td>
</tr>
<tr>
<td>0.80</td>
<td>0.000397</td>
<td>0.000372</td>
<td>2.4645e-05</td>
</tr>
<tr>
<td>1.00</td>
<td>0.000054</td>
<td>0.000052</td>
<td>1.7941e-06</td>
</tr>
</tbody>
</table>

This output is equivalent to the output from the preceding MATLAB (Section 4.1) and C (Section 4.2); i.e., the error tolerances in \texttt{intpar} are exceeded because of the spatial discretization errors. We now proceed to the Fortran programming of the linear PDE.
4.4 Programming in Fortran

Again, since main Program 2.4.1 is unchanged, it is not listed here. \textit{intpar}, \textit{par}, \textit{initial}, \textit{deriv}, and \textit{fprint} are listed below:

```fortran
subroutine intpar(neqn,nout,nsteps,t0,tf,abserr,relerr)
C
C Subroutine intpar sets the parameters to control the
C integration of the linear PDE problem
C
C Double precision coding is used
    implicit double precision(a-h,o-z)
C
C Number of ODEs
    neqn=21
C
C Number of output points
    nout=6
C
C Maximum number of steps in the interval t0 to tf
    nsteps=250
C
C Initial, final values of the independent variable
    t0=0.0d0
    tf=0.2d0
C
C Error tolerances
    abserr=1.0d-05
    relerr=1.0d-05
    return
C
C End of intpar
end

subroutine par(xl,xu,pi)
C
C Subroutine par sets the parameters for the linear
C PDE problem
C
C Double precision coding is used
    implicit double precision(a-h,o-z)
C
C Problem parameters
```

Copyright © 2004 by Chapman & Hall/CRC
subroutine inital(neqn,t,u0)
C
C Subroutine inital sets the initial condition vector
C for the linear PDE problem
C
C Double precision coding is used
   implicit double precision(a-h,o-z)
C
C Size the arrays
   dimension u0(neqn)
C
C Problem parameters
   call par(xl,xu,pi)
C
C Initial condition
   do i=1,neqn
      x=xl+dfloat(i-1)/dfloat(neqn-1)*(xu-xl)
      u0(i)=dsin(pi*x)
   end do
return
C
C End of inital
end

subroutine derv(neqn,t,u,ut)
C
C Subroutine derv computes the derivative vector
C of the linear PDE problem
C
C Double precision coding is used
   implicit double precision(a-h,l,o-z)
C
C Size the arrays
   dimension u(neqn), ut(neqn)
C
C Problem parameters
     call par(xl,xu,pi)
C
C BC at x = 0
     ut(1)=0.0d0
C
C BC at x = 1
     ut(neqn)=0.0d0
C
C Interior points
     dx=(xu-xl)/dfloat(neqn-1)
     dxs=dx*dx
     do i=2,neqn-1
       ut(i)=(u(i+1)-2.0d0*u(i)+u(i-1))/dxs
     end do
     return
C
C End of derv
end

subroutine fprint(no,ncase,neqn,t,u)
C
C Subroutine fprint displays the numerical and
C exact solutions to the linear PDE problem
C
C Double precision coding is used
     implicit double precision(a-h,l,o-z)
C
C Size the arrays
     dimension u(neqn)
C
C Problem parameters
     call par(xl,xu,pi)
C
C Print a heading for the solution at t = 0
     if(t.le.0.0d0)then
       C
       C Label for ODE integrator
       C
       C Fixed step modfied Euler
       if(ncase.eq.1)then
         write(no,11)
         11 format(//,' euler2a integrator')
     end if
     return
end
C Variable step modified Euler
else if(ncase.eq.2)then
  write(no,12)
  format(//,' euler2b integrator')
C
C Fixed step classical fourth order RK
else if(ncase.eq.3)then
  write(no,13)
  format(//,' rkc4a integrator')
C
C Variable step classical fourth order RK
else if(ncase.eq.4)then
  write(no,14)
  format(//,' rkc4b integrator')
C
C Fixed step RK Fehlberg 45
else if(ncase.eq.5)then
  write(no,15)
  format(//,' rkf45a integrator')
C
C Variable step RK Fehlberg 45
else if(ncase.eq.6)then
  write(no,16)
  format(//,' rkf45b integrator')
end if
C
C Heading
  write(no,4)ncase,neqn
  4 format(//,' ncase = ',i1,' neqn = ',i2)
  write(no,2)
  2 format(//,4x,'t',4x,'u1(num)',5x,'u1(ex)',8x,'diff1',/)
C
C End of t = 0 heading
end if
C
C Numerical and analytical solution output
C
C Midpoint value of x
  x=(xu-xl)/2.0d0
C
C Analytical solution at midpoint
  ue=dexp(-pi*pi*t)*dsin(pi*x)
C
C Grid index of midpoint
  im=dfloat((neqn+1)/2)
C Display the numerical and exact solutions, and their difference
write(no,3) t, u(im), u(ex), u(im) - u(ex)
3 format(f5.2,2f11.6,e13.4)
return
C End of fprintf
end

Program 4.4.1
intpar, par, inital, der0, and fprintf for the solution of Equations 4.1 to 4.4

The output from the routines of Program 4.4.1 is listed below:

euler2a integrator
ncase = 1  neqn = 21

<table>
<thead>
<tr>
<th>t</th>
<th>u1(num)</th>
<th>u1(ex)</th>
<th>diff1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.0000E+00</td>
</tr>
<tr>
<td>0.20</td>
<td>0.139478</td>
<td>0.138911</td>
<td>0.5673E-03</td>
</tr>
<tr>
<td>0.40</td>
<td>0.019454</td>
<td>0.019296</td>
<td>0.1579E-03</td>
</tr>
<tr>
<td>0.60</td>
<td>0.002713</td>
<td>0.002680</td>
<td>0.3298E-04</td>
</tr>
<tr>
<td>0.80</td>
<td>0.000378</td>
<td>0.000372</td>
<td>0.6120E-05</td>
</tr>
<tr>
<td>1.00</td>
<td>0.000053</td>
<td>0.000052</td>
<td>0.1065E-05</td>
</tr>
</tbody>
</table>

euler2b integrator
ncase = 2  neqn = 21

<table>
<thead>
<tr>
<th>t</th>
<th>u1(num)</th>
<th>u1(ex)</th>
<th>diff1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.0000E+00</td>
</tr>
<tr>
<td>0.20</td>
<td>0.139530</td>
<td>0.138911</td>
<td>0.6189E-03</td>
</tr>
<tr>
<td>0.40</td>
<td>0.019464</td>
<td>0.019296</td>
<td>0.1678E-03</td>
</tr>
<tr>
<td>0.60</td>
<td>0.002713</td>
<td>0.002680</td>
<td>0.3661E-04</td>
</tr>
<tr>
<td>0.80</td>
<td>0.000379</td>
<td>0.000372</td>
<td>0.6835E-05</td>
</tr>
<tr>
<td>1.00</td>
<td>0.000057</td>
<td>0.000052</td>
<td>0.5205E-05</td>
</tr>
</tbody>
</table>
rkc4a integrator

ncase = 3  neqn = 21

<table>
<thead>
<tr>
<th>t</th>
<th>u1(num)</th>
<th>u1(ex)</th>
<th>diff1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.0000E+00</td>
</tr>
<tr>
<td>0.20</td>
<td>0.139476</td>
<td>0.138911</td>
<td>0.5645E-03</td>
</tr>
<tr>
<td>0.40</td>
<td>0.019453</td>
<td>0.019296</td>
<td>0.1571E-03</td>
</tr>
<tr>
<td>0.60</td>
<td>0.002713</td>
<td>0.002680</td>
<td>0.3281E-04</td>
</tr>
<tr>
<td>0.80</td>
<td>0.000378</td>
<td>0.000372</td>
<td>0.6089E-05</td>
</tr>
<tr>
<td>1.00</td>
<td>0.000053</td>
<td>0.000052</td>
<td>0.1059E-05</td>
</tr>
</tbody>
</table>

rkc4b integrator

ncase = 4  neqn = 21

<table>
<thead>
<tr>
<th>t</th>
<th>u1(num)</th>
<th>u1(ex)</th>
<th>diff1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.0000E+00</td>
</tr>
<tr>
<td>0.20</td>
<td>0.139472</td>
<td>0.138911</td>
<td>0.5605E-03</td>
</tr>
<tr>
<td>0.40</td>
<td>0.019453</td>
<td>0.019296</td>
<td>0.1568E-03</td>
</tr>
<tr>
<td>0.60</td>
<td>0.002712</td>
<td>0.002680</td>
<td>0.3165E-04</td>
</tr>
<tr>
<td>0.80</td>
<td>0.000378</td>
<td>0.000372</td>
<td>0.5946E-05</td>
</tr>
<tr>
<td>1.00</td>
<td>0.000052</td>
<td>0.000052</td>
<td>-0.1941E-06</td>
</tr>
</tbody>
</table>

rkf45a integrator

ncase = 5  neqn = 21

<table>
<thead>
<tr>
<th>t</th>
<th>u1(num)</th>
<th>u1(ex)</th>
<th>diff1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.0000E+00</td>
</tr>
<tr>
<td>0.20</td>
<td>0.139476</td>
<td>0.138911</td>
<td>0.5645E-03</td>
</tr>
<tr>
<td>0.40</td>
<td>0.019453</td>
<td>0.019296</td>
<td>0.1571E-03</td>
</tr>
<tr>
<td>0.60</td>
<td>0.002713</td>
<td>0.002680</td>
<td>0.3281E-04</td>
</tr>
<tr>
<td>0.80</td>
<td>0.000378</td>
<td>0.000372</td>
<td>0.6089E-05</td>
</tr>
<tr>
<td>1.00</td>
<td>0.000053</td>
<td>0.000052</td>
<td>0.1059E-05</td>
</tr>
</tbody>
</table>
rkf45b integrator

ncase = 6  neqn = 21

t  u1(num)  u1(ex)  diff1
0.00  1.000000  1.000000  0.0000E+00
0.20  0.139476  0.138911  0.5644E-03
0.40  0.019446  0.019296  0.1498E-03
0.60  0.002715  0.002680  0.3457E-04
0.80  0.000380  0.000372  0.7272E-05
1.00  0.000053  0.000052  0.1161E-05

This output is equivalent to the output from the preceding MATLAB (Section 4.1), C (Section 4.2), and C++ (Section 4.3); i.e., the error tolerances in intpar are exceeded because of the spatial discretization errors. We now proceed to the Java programming of the linear PDE.

4.5 Programming in Java

Again, since main Program 2.5.1 is unchanged in the linear PDE problem, it is not listed here. An interface routine is listed below:

```java
/*
   Define the common (global) variables for the linear PDE integration
*/

package mol;

public class MOL
{
  /* Maximum (default) number of ODEs */
  public int SIZE=500;

  /* Variables for ODE integration */
  public int neqn, nout, nsteps;
  public double t0, tf, abserr, relerr;
  public double u[], u0[], e[];
```
/* Parameters for PDE */

public double pi, xl, xu, dx, dxs;

public MOL()
{
}

Program 4.5.1
Interface routine for the solution of Equations 4.1 to 4.4

intpar, par, initial, derv, and fprint are listed next:

/* This file is a member of the package mol */
package mol;

import mol.MOL;
import java.math.*;
import java.io.*;
import java.text.*;

public class DEF extends MOL implements pdelininterface
{
    public DEF()
    {
        /* Integration parameters */
        this.intpar();

        /* Declare arrays */
        u0=new double[SIZE];
        u=new double[SIZE];
        e=new double[SIZE];

        /* Problem parameters */
        this.par();

        /* Initial condition vector */
        this.initial();
    }

    public void intpar()
    {
        /* Function intpar sets the parameters to control the integration of the linear PDE system */
    }

    Copyright © 2004 by Chapman & Hall/CRC
{ /* Number of ODEs */
  neqn=21;

  /* Size of arrays in MOL library */
  SIZE=neqn+1;

  /* Number of output points */
  nout=6;

  /* Maximum number of steps in the interval t0 to tf */
  nsteps=250;

  /* Initial, final values of the independent variable */
  t0=0.0;
  tf=0.2;

  /* Error tolerances */
  abserr=Math.pow(10.0,-5.0);
  relerr=Math.pow(10.0,-5.0);

  /* End of intpar */
}

public void par()

  /* Function par sets the parameters for the linear
   PDE problem */
  {
    pi=4.0*Math.atan(1.0);
    xl=0.0;
    xu=1.0;
    dx=(xu-xl)/(neqn-1);
    dxs=dx*dx;

    /* End of par */
  }

public void initial()

  /* Function initial sets the initial condition vector
   for the linear PDE problem */
  {
/ * Type variables */
double x;
int i;

/* Initial condition */
for(i=1;i<=neqn;i++)
{
  x=xl+(double)(i-1)/(neqn-1)*(xu-xl);
  u0[i]=Math.sin(pi*x);
}

/* End of initial */
}

public void derv(double ut[], double t, double u[])

/* Function derv computes the derivative vector
of the linear PDE problem */
{
  /* Type variables */
  int i;

  /* Problem parameters */
  par();

  /* BC at x = xl */
  ut[1]=0.0;

  /* BC at x = xu */
  ut[neqn]=0.0;

  /* Derivative vector */
  for(i=2;i<=(neqn-1);i++)
  {
    /* Approximation to the PDE */
    ut[i]=(u[i+1]-2.0*u[i]+u[i-1])/dxs;
  }

  /* End of derv */
}

public void fprint(PrintWriter f, int ncase, int neqn,
double t, double u[])
/ * Function fprint displays the numerical and exact
  solutions to the linear PDE problem */
{
  /* Type variables */
  double x, ue;
  int im;

  /* Print a heading for the solution at t = 0 */
  if(t<=0.0)
  {
    /* Label for ODE integrator */
    switch(ncase)
    {
      /*Fixed step modified Euler */
      case 1:
        f.println("\n\n euler2a integrator\n");
        break;

      /* Variable step modified Euler */
      case 2:
        f.println("\n\n euler2b integrator\n");
        break;

      /* Fixed step classical fourth order RK */
      case 3:
        f.println("\n\n rkc4a integrator\n");
        break;

      /* Variable step classical fourth order RK */
      case 4:
        f.println("\n\n rkc4b integrator\n");
        break;

      /* Fixed step RK Fehlberg 45 */
      case 5:
        f.println("\n\n rkf45a integrator\n");
        break;

      /* Variable step RK Fehlberg 45 */
      case 6:
        f.println("\n\n rkf45b integrator\n");
        break;
    }
  }
/* Heading */
f.println(" ncase = "+ncase+" neqn = "+neqn+"\n");
f.println(" t u(num) u(exact) diff");
/* End of t = 0 heading */
}

/* Numerical and analytical solution output */

/* Midpoint value of x */
x=(xu-xl)/2.0;

/* Analytical solution at midpoint */
ue=Math.exp(-pi*pi*t)*Math.sin(pi*x);

/* Grid index of midpoint */
im=(neqn+1)/2;

/* Display format for floating numbers */
DecimalFormat df1=new DecimalFormat("0.00");
DecimalFormat df2=new DecimalFormat("0.0000000000");

/* Display the numerical and exact solutions,
and their difference */
f.println(df1.format(t)+"\t"+df2.format(u[im])
+"\t"+df2.format(ue)+"\t"+df2.format(u[im]-ue));

/* End of fprintf */
}

/* End of DEF */
}

Program 4.5.2
intpar, par, initial, derv, and fprintf for the solution of Equations 4.1 to 4.4

The output from the preceding routines is
euler2a integrator
ncase = 1  neqn = 21

t   u(num)   u(exact)   diff
0.00 1.00000000000 1.00000000000 0.00000000000
0.20 0.13947847360 0.13891113310 0.00056734050
**euler2b integrator**

ncase = 2  neqn = 21

<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.0000000000</td>
<td>1.0000000000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>0.20</td>
<td>0.13953000624</td>
<td>0.1389111331</td>
<td>0.0006189292</td>
</tr>
<tr>
<td>0.40</td>
<td>0.0194641053</td>
<td>0.0192963029</td>
<td>0.0001678024</td>
</tr>
<tr>
<td>0.60</td>
<td>0.0027170826</td>
<td>0.0026804713</td>
<td>0.0000366113</td>
</tr>
<tr>
<td>0.80</td>
<td>0.0003791827</td>
<td>0.0003723473</td>
<td>0.0000068354</td>
</tr>
<tr>
<td>1.00</td>
<td>0.0000569278</td>
<td>0.0000517232</td>
<td>0.0000052046</td>
</tr>
</tbody>
</table>

**rkc4a integrator**

ncase = 3  neqn = 21

<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.0000000000</td>
<td>1.0000000000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>0.20</td>
<td>0.1394756138</td>
<td>0.1389111331</td>
<td>0.0005644806</td>
</tr>
<tr>
<td>0.40</td>
<td>0.0194534468</td>
<td>0.0192963029</td>
<td>0.0001571439</td>
</tr>
<tr>
<td>0.60</td>
<td>0.0027132814</td>
<td>0.0026804713</td>
<td>0.0000328101</td>
</tr>
<tr>
<td>0.80</td>
<td>0.0003784366</td>
<td>0.0003723473</td>
<td>0.0000060893</td>
</tr>
<tr>
<td>1.00</td>
<td>0.0000569278</td>
<td>0.0000517232</td>
<td>0.0000052046</td>
</tr>
</tbody>
</table>

**rkc4b integrator**

ncase = 4  neqn = 21

<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.0000000000</td>
<td>1.0000000000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>0.20</td>
<td>0.1394715959</td>
<td>0.1389111331</td>
<td>0.0005604628</td>
</tr>
<tr>
<td>0.40</td>
<td>0.0194530717</td>
<td>0.0192963029</td>
<td>0.0001567688</td>
</tr>
<tr>
<td>0.60</td>
<td>0.0027121171</td>
<td>0.0026804713</td>
<td>0.0000316458</td>
</tr>
<tr>
<td>0.80</td>
<td>0.0003782931</td>
<td>0.0003723473</td>
<td>0.0000059458</td>
</tr>
<tr>
<td>1.00</td>
<td>0.0000515291</td>
<td>0.0000517232</td>
<td>-0.000001941</td>
</tr>
</tbody>
</table>
rkf45a integrator

ncase = 5  neqn = 21

<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.0000000000</td>
<td>1.0000000000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>0.20</td>
<td>0.1394756138</td>
<td>0.1389111331</td>
<td>0.0005644806</td>
</tr>
<tr>
<td>0.40</td>
<td>0.0194534468</td>
<td>0.0192963029</td>
<td>0.0001571439</td>
</tr>
<tr>
<td>0.60</td>
<td>0.0027132814</td>
<td>0.0026804713</td>
<td>0.0000328101</td>
</tr>
<tr>
<td>0.80</td>
<td>0.0003784366</td>
<td>0.0003723473</td>
<td>0.000060893</td>
</tr>
<tr>
<td>1.00</td>
<td>0.0000527827</td>
<td>0.0000517232</td>
<td>0.000010595</td>
</tr>
</tbody>
</table>

rkf45b integrator

ncase = 6  neqn = 21

<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.0000000000</td>
<td>1.0000000000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>0.20</td>
<td>0.1394755775</td>
<td>0.1389111331</td>
<td>0.0005644444</td>
</tr>
<tr>
<td>0.40</td>
<td>0.0194461297</td>
<td>0.0192963029</td>
<td>0.0001498268</td>
</tr>
<tr>
<td>0.60</td>
<td>0.0027150369</td>
<td>0.0026804713</td>
<td>0.0000345656</td>
</tr>
<tr>
<td>0.80</td>
<td>0.0003796192</td>
<td>0.0003723473</td>
<td>0.0000072719</td>
</tr>
<tr>
<td>1.00</td>
<td>0.0000528840</td>
<td>0.0000517232</td>
<td>0.000011608</td>
</tr>
</tbody>
</table>

This output is equivalent to the output from the preceding MATLAB (Section 4.1), C (Section 4.2), C++ (Section 4.3), and Fortran (Section 4.4); i.e., the error tolerances in intpar are exceeded because of the spatial discretization errors. We now proceed to the Maple programming of the linear PDE.

4.6 Programming in Maple

Since main Program 4.6.1 (and subordinate routines) accesses specific files by read statements, it is listed first:

> restart:

> read "c:\\odelib\\maple\\pdelin\\pdelin.txt";

> pdelin();

Program 4.6.1

Maple main program pdelin.mws for the numerical integration of Equations 4.1 to 4.4
pdelin:=proc()
# # Main program pdelin computes the numerical
# solution to the linear PDE system by one of
# six integrators
# # Type variables
global neqn, nout, nsteps, t0, tf, abserr, relerr:
local u0, u, tp, ncase, i, j:
# # Step through six integrators
for ncase from 1 to 6 do
# # Integration parameters
read "c:\odelib\maple\pdelin\intpar.txt":
intpar():
# # Size arrays
u0:=array(1..neqn): u:=array(1..neqn):
# # Initial condition vector
read "c:\odelib\maple\pdelin\inital.txt":
inital(neqn,t0,u0):
# # Output interval
tp:=tf-t0:
# # Compute solution at nout output points
for j from 1 to nout do
# # Print current solution
read "c:\odelib\maple\pdelin\fprint.txt":
fprint(ncase,neqn,t0,u0):
# # Fixed step modified Euler integrator
if (ncase = 1) then
read "c:\odelib\maple\pdelin\euler2a.txt":
euler2a(neqn,t0,tf,u0,nsteps,u):
end if:
# # Variable step modified Euler integrator
if (ncase = 2) then
read "c:\odelib\maple\pdelin\euler2b.txt":
euler2b(neqn,t0,tf,u0,nsteps,abserr,relerr,u):
end if:
# Fixed step classical fourth order RK integrator
if (ncase = 3) then
    read "c:\odelib\maple\pdelin\rkc4a.txt":
    rkc4a(neqn,t0,tf,u0,nsteps,u):
end if:
#
# Variable step classical fourth order RK integrator
if (ncase = 4) then
    read "c:\odelib\maple\pdelin\rkc4b.txt":
    rkc4b(neqn,t0,tf,u0,nsteps,abserr,relerr,u):
end if:
#
# Fixed step RK Fehlberg (RKF45) integrator
if (ncase = 5) then
    read "c:\odelib\maple\pdelin\rkf45a.txt":
    rkf45a(neqn,t0,tf,u0,nsteps,u):
end if:
#
# Variable step RK Fehlberg (RKF45) integrator
if (ncase = 6) then
    read "c:\odelib\maple\pdelin\rkf45b.txt":
    rkf45b(neqn,t0,tf,u0,nsteps,abserr,relerr,u):
end if:
#
# Advance solution
    t0:=tf;
    tf:=tf+tp:
    for i from 1 to neqn do
        u0[i]:=u[i]:
    end do:
#
# Next output
end do:
#
# Next integrator
end do:
#
# End of pdelin
end:

Program 4.6.2
Maple main program pdelin.txt for the numerical integration of Equations 4.1 to 4.4
Note the reference to specific files by read statements, e.g.,

```maple
# # Initial condition vector
read "c:\odelib\maple\pdelin\initial.txt":
inital(neqn,t0,u0):
```

`intpar, inital, derv, and fprint` are listed below:

```maple
intpar:=proc()
# # Function intpar sets the parameters to control the
# integration of the linear PDE problem
# # Type variables
  global neqn, nout, nsteps, t0, tf, abserr, relerr:
# # Number of first order ODEs
  neqn:=21:
# # Number of output points
  nout:=6:
# # Maximum number of steps in the interval t0 to tf
  nsteps:=250:
# # Initial, final values of independent variable
  t0:=0.0:
tf:=0.2:
# # Error tolerances
  abserr:=1.0e-05:
  relerr:=1.0e-05:
# # End of intpar
end:

inital:=proc(neqn,t,u0)
# # Function inital sets the initial condition vector
# of the linear PDE problem
#```
# Type variables
local xl, xu, pi, x, i:
#
# Problem parameters
xl:=0.0:
xu:=1.0:
#
# Initial condition
pi:=evalf(Pi):
for i from 1 to neqn do
x:=xl+(i-1)/(neqn-1)*(xu-xl):
u0[i]:=sin(pi*x):
end do:
#
# End of initial
end:

derv:=proc(neqn,t,u,ut)
#
# Function derv computes the derivative vector
# of the linear PDE problem
#
# Type variables
local xl, xu, dx, dxs, i:
#
# Problem parameters
xl:=0.0:
xu:=1.0:
#
# BC at x = 0
ut[1]:=0.0:
#
# BC at x = 1
ut[neqn]:=0.0:
#
# Interior points
dx:=(xu-xl)/(neqn-1):
dxs:=dx*dx:
for i from 2 to neqn-1 do
ut[i]:=(u[i+1]-2.0*u[i]+u[i-1])/dxs:
end do:
#
# End of derv
end:
fprint:=proc(ncase,neqn,t,u)
#
# Function fprint displays the numerical and exact
# solutions to the linear PDE problem
#
# Type variables
local xl, xu, x, pi, ue, im:
#
# Declare global variables
global nsteps:
#
# Problem parameters
xl:=0.0:
xu:=1.0:
#
# Print a heading for the solution at \( t = 0 \)
if (t <= 0.0) then
#
# Label for ODE integrator
#
# Fixed step modified Euler
if (ncase = 1) then
  printf(`

  euler2a integrator

`);
#
# Variable step modified Euler
elif (ncase = 2) then
  printf(`

  euler2b integrator

`);
#
# Fixed step classical fourth order RK
elif (ncase = 3) then
  printf(`

  rkc4a integrator

`);
#
# Variable step classical fourth order RK
elif (ncase = 4) then
  printf(`

  rkc4b integrator

`);
#
# Fixed step RK Fehlberg 45
elif (ncase = 5) then
  printf(`

  rkf45a integrator

`);
#
# Variable step RK Fehlberg 45
elif (ncase = 6) then
  printf(`

  rkf45b integrator

`);
end if:
## Heading

printf(` ncase = %2d  neqn = %2d  nsteps = %3d 

`,
ncase,neqn,nsteps);
printf(` t  u(num)  u(exact)  diff
`);
#
# End of t = 0 heading
end if:
#
# Numerical and analytical solution output
#
# Midpoint value of x
x:=(xu-xl)/2.0:
#
# Analytical solution at midpoint
pi:=evalf(Pi):
ue:=exp(-pi*pi*t)*sin(pi*x):
#
# Grid index of midpoint
im:=round((neqn+1)/2):
#
# Display the numerical and exact solutions,
# and their difference
printf(`%5.2f %11.6f %11.6f %13.4e
`,t,u[im],ue,u[im]-ue);
#
# End of fprint
end:

**Program 4.6.3**

`intpar`, `initial`, `deriv`, and `fprint` for the solution of Equations 4.1 to 4.4

Note the use of the Maple in-line value of π in `initial`, `deriv`, and `fprint`, e.g.,

## Initial condition

pi:=evalf(Pi):
for i from 1 to neqn do
  x:=xl+(i-1)/(neqn-1)*(xu-xl):
  u0[i]:=sin(pi*x):
end do:

Pi is a symbolic value of π which is then converted to a numerical value, `evalf(Pi)`, for use in subsequent calculations.
The output from the preceding routines is listed below:

### euler2a integrator

<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.0000e-01</td>
</tr>
<tr>
<td>0.20</td>
<td>.139478</td>
<td>.138911</td>
<td>5.6734e-04</td>
</tr>
<tr>
<td>0.40</td>
<td>.019454</td>
<td>.019296</td>
<td>1.5794e-04</td>
</tr>
<tr>
<td>0.60</td>
<td>.002713</td>
<td>.002680</td>
<td>3.2977e-05</td>
</tr>
<tr>
<td>0.80</td>
<td>.000378</td>
<td>.000372</td>
<td>6.1201e-06</td>
</tr>
<tr>
<td>1.00</td>
<td>.000053</td>
<td>.000052</td>
<td>1.0647e-06</td>
</tr>
</tbody>
</table>

### euler2b integrator

<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.0000e-01</td>
</tr>
<tr>
<td>0.20</td>
<td>.139530</td>
<td>.138911</td>
<td>6.1893e-04</td>
</tr>
<tr>
<td>0.40</td>
<td>.019462</td>
<td>.019296</td>
<td>1.6544e-04</td>
</tr>
<tr>
<td>0.60</td>
<td>.002712</td>
<td>.002680</td>
<td>3.1091e-05</td>
</tr>
<tr>
<td>0.80</td>
<td>.000378</td>
<td>.000372</td>
<td>5.6249e-06</td>
</tr>
<tr>
<td>1.00</td>
<td>.000049</td>
<td>.000052</td>
<td>-3.1368e-06</td>
</tr>
</tbody>
</table>

### rkc4a integrator

<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.0000e-01</td>
</tr>
<tr>
<td>0.20</td>
<td>.139476</td>
<td>.138911</td>
<td>5.6448e-04</td>
</tr>
<tr>
<td>0.40</td>
<td>.019453</td>
<td>.019296</td>
<td>1.5714e-04</td>
</tr>
<tr>
<td>0.60</td>
<td>.002713</td>
<td>.002680</td>
<td>3.2810e-05</td>
</tr>
<tr>
<td>0.80</td>
<td>.000378</td>
<td>.000372</td>
<td>6.0891e-06</td>
</tr>
<tr>
<td>1.00</td>
<td>.000053</td>
<td>.000052</td>
<td>1.0593e-06</td>
</tr>
</tbody>
</table>
### rkc4b integrator

**ncase = 4  neqn = 21  nsteps = 250**

<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.0000e-01</td>
</tr>
<tr>
<td>.20</td>
<td>.139473</td>
<td>.138911</td>
<td>5.6189e-04</td>
</tr>
<tr>
<td>.40</td>
<td>.019453</td>
<td>.019296</td>
<td>1.5690e-04</td>
</tr>
<tr>
<td>.60</td>
<td>.002713</td>
<td>.002680</td>
<td>3.2058e-05</td>
</tr>
<tr>
<td>.80</td>
<td>.000376</td>
<td>.000372</td>
<td>3.7570e-06</td>
</tr>
<tr>
<td>1.00</td>
<td>.000053</td>
<td>.000052</td>
<td>8.4148e-07</td>
</tr>
</tbody>
</table>

### rkf45a integrator

**ncase = 5  neqn = 21  nsteps = 250**

<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.0000e-01</td>
</tr>
<tr>
<td>.20</td>
<td>.139476</td>
<td>.138911</td>
<td>5.6448e-04</td>
</tr>
<tr>
<td>.40</td>
<td>.019453</td>
<td>.019296</td>
<td>1.5714e-04</td>
</tr>
<tr>
<td>.60</td>
<td>.002713</td>
<td>.002680</td>
<td>3.2810e-05</td>
</tr>
<tr>
<td>.80</td>
<td>.000376</td>
<td>.000372</td>
<td>6.0891e-06</td>
</tr>
<tr>
<td>1.00</td>
<td>.000053</td>
<td>.000052</td>
<td>1.0593e-06</td>
</tr>
</tbody>
</table>

### rkf45b integrator

**ncase = 6  neqn = 21  nsteps = 250**

<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.0000e-01</td>
</tr>
<tr>
<td>.20</td>
<td>.139476</td>
<td>.138911</td>
<td>5.6448e-04</td>
</tr>
<tr>
<td>.40</td>
<td>.019449</td>
<td>.019296</td>
<td>1.5311e-04</td>
</tr>
<tr>
<td>.60</td>
<td>.002714</td>
<td>.002680</td>
<td>3.3779e-05</td>
</tr>
<tr>
<td>.80</td>
<td>.000379</td>
<td>.000372</td>
<td>6.1720e-06</td>
</tr>
<tr>
<td>1.00</td>
<td>.000053</td>
<td>.000052</td>
<td>9.4329e-07</td>
</tr>
</tbody>
</table>

This output is equivalent to the output from the preceding MATLAB (Section 4.1), C (Section 4.2), C++ (Section 4.3), Fortran (Section 4.4), and Java.
(Section 4.5), i.e., the error tolerances in intpar are exceeded because of the spatial discretization errors.

This completes the discussion of the linear PDE problem programmed in the six languages. Basically, we have observed that PDEs can also be integrated using the methods for ODEs discussed in Chapters 2 and 3. The additional step is the algebraic approximation of the spatial (boundary value) derivatives in the PDEs, including the boundary conditions, so that the PDEs are replaced by an approximating system of initial values ODEs. This procedure is the essence of the method of lines.

We now consider the final problem, a nonlinear PDE, in the remaining chapter. Again, as in the case of the preceding linear PDE, the techniques for ODEs in Chapters 2 and 3 can also be applied to the nonlinear PDE problem.

References

The concluding PDE example is based on a physical system comprising a section of steel with a layer of external insulation, as depicted in Figure 5.1. The external surface of the insulation is heated by radiation from a flame, and the basic question to be answered then is how the temperature of the steel changes with time due to the radiative heating (the insulation is designed to prevent rapid heating of the steel).

To analyze this system, we first realize that the temperature varies across the insulation as well as with time. Thus, we have a problem in time and space, as we did with the PDE of Chapter 4, Equation 4.1, and we therefore will again use a PDE to analyze the system of Figure 5.1. Since we can generally not model a physical system in complete detail, we will make some simplifying assumptions so that the analysis becomes tractable, yet we expect that the resulting mathematical model will still capture the essential features (the first-order effects) of the problem. Here then is a list of assumptions:

1. The conduction of heat along the insulation is significant in only one direction, the smallest dimension (transverse to the cross section); thus the insulation “thickness” is assumed to be much smaller than its other two (longitudinal) dimensions. Again we use $x$ and $t$ to denote position (in the insulation) and time.

2. Consequently, the temperature variation is also only one dimensional (in the same direction as the heat conduction).

3. Because the steel has a much higher thermal conductivity than the insulation, the variation of temperature in the steel with position will be much smaller than the spatial temperature variation in the insulation. Thus, we will assume that the steel is at a single, uniform temperature (which can vary with time).

4. Since the steel temperature does not vary spatially, we have only one independent variable for the steel, time, and the steel is therefore modeled by an ODE in time.
5. The physical properties of the insulation and steel are constant (do not vary with temperature). This may be a reasonable assumption, or may be sufficiently accurate if we use properties averaged over the temperatures that the insulation and steel experience. If this is not a reasonable assumption, the properties can be computed as a function of temperature and used for each $x$ and $t$.

The temperature profiles that we will be calculating are illustrated in Figure 5.2. Note that the system (insulation and steel) starts at a temperature of $25^\circ\text{C} = 298\text{ K}$. The left face of the insulation ($z = 0$) is then subjected to a flame at a temperature of $T_f = 2000\text{ K}$. The temperatures in the insulation and the steel then begin to rise, and if the solution is computed to sufficiently large time $t$, the entire system would come to a uniform temperature of $2000\text{ K}$. Some intermediate temperature profiles (for finite $t$) are illustrated.
FIGURE 5.3
Transient heating of composite steel–insulation system.

By energy conservation
on a finite volume
accumulation = input - output

$$A\Delta x \rho C_p \frac{dT_i}{dt} = A(q_{in1/2} - q_{out1/2})$$

By a FD approximation to Fourier’s first law

$$q_{in1/2} = -k\frac{(T_{in1} - T_i)}{\Delta x} \quad q_{out1/2} = -k\frac{(T_i - T_{out1})}{\Delta x}$$

$$A\Delta x \rho C_p \frac{dT_i}{dt} = A\left[k\left(\frac{T_{in1} - T_i}{\Delta x}\right) - k\left(\frac{T_i - T_{out1}}{\Delta x}\right)\right]$$

$$\frac{dT_i}{dt} = \frac{k}{\rho C_p} \left[\frac{T_{in1} - T_i}{\Delta x}\right] - \frac{k}{\rho C_p} \left[\frac{T_i - T_{out1}}{\Delta x}\right]$$

For $\Delta x \to 0$, Fourier’s second law:

$$\frac{dT}{dt} = \alpha \frac{\partial^2 T}{\partial x^2}$$

FIGURE 5.4
Energy balance on the incremental volume of insulation.

in Figure 5.2, and in more detail in Figure 5.3. Note in Figure 5.3 the definition of a finite volume of the insulation of thickness $\Delta z$. The mathematical model for the insulation is based on an energy balance for this incremental volume. The derivation of the mathematical model is outlined in Figure 5.4.
The final result of the energy balance, in the limit $\Delta z \rightarrow 0$, is again Fourier’s second law, which we now state in terms of $u$ for temperature (rather than $T$):

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}$$

(5.1)

The initial and boundary conditions for Equation 5.1 taken as

$$u(x, 0) = u_0$$

(5.2)

$$-k \frac{\partial u(0, t)}{\partial x} = \sigma (a u_a^4 - e u^4(0, t))$$

(5.3)

$$L_s \rho_s C_{ps} \frac{du_s}{dt} = -k \frac{\partial u(L, t)}{\partial x}$$

(5.4)

where, in SI (MKS) units,

$u =$ insulation temperature, K

t = time, s

x = position in the insulation, m

$u_a =$ ambient (flame) temperature, K

$\alpha =$ insulation thermal diffusivity, m$^2$/s

$k =$ insulation thermal conductivity, J · m$^2$/s · K

$\sigma =$ Stefan–Boltzmann constant

$$= 5.67 \times 10^{-8} \frac{J}{s \cdot m^2 \cdot K^4}$$

$L =$ insulation half thickness, m

a = absorptivity

$e =$ emissivity

$u_s =$ steel temperature, K

$e =$ emissivity

$L_s =$ steel half thickness, m

$\rho_s =$ steel density, kg/m$^3$

$C_{ps} =$ steel specific heat, J/kg · K

Equations 5.3 and 5.4 are also energy balances, but applied at the boundaries of the insulation and steel. For Equation 5.3, we have the following:

• $-k \frac{\partial u(0, t)}{\partial x}$, the rate of heat conduction at the left surface of the insulation (at $x = 0$) according to Fourier’s first law. This term is actually a flux (a rate of heat conduction per unit area of insulation) with the units J/m$^2$ · s.

• $\sigma (a u_a^4 - e u^4(0, t))$, the rate of radiative heat transfer from the flame to the insulation surface according to the Stefan–Boltzmann law. Again, this is a flux with the units J/m$^2$ · s. Note that the dependent variable (the insulation temperature) is raised to the fourth power, $u^4(0, t)$. This is a nonlinear term in the model. Thus, although Equation 5.1 is linear, this nonlinear BC is the reason the model is considered nonlinear (and, also, the reason an analytical solution is precluded).
• Equation 5.3 is therefore an energy balance that equates the rate of radiative heat transfer to the rate of heat conduction at the surface of the insulation.

For Equation 5.4, we have an energy balance on the steel:

• \( L_s \rho_s C_{ps} \frac{d u_s}{d t} \), the rate of accumulation of energy within the steel (with thickness \( L_s \)). This term again has the units of J/m\(^2\) · s.

• \(-k \frac{\partial u(L, t)}{\partial x}\), the rate of heat conduction from the right face of the insulation (at \( x = L \)) to the steel. Since this is a flux, it has the units of J/m\(^2\) · s.

• Equation 5.4 is therefore an energy balance that equates the rate of heat transfer from the insulation to the rate of accumulation of energy within the steel.

• Equation 5.4 is actually an ODE in time. Therefore, the model is a mixed ODE/PDE system. This feature does not present any particular difficulty in computing a numerical solution since the PDE, Equation 5.1, will be converted to a system of \( n = 21 \) ODEs, as before in Chapter 4. Thus, we will merely add Equation 5.4 as another ODE for a system of 22 ODEs.

• This last point illustrates an important advantage of the method of lines, i.e., it can easily accommodate mixed ODE/PDE systems.

• Since Equation 5.4 is an ODE, it requires an initial condition, which we take as

\[
 u_s(0) = u_0
\]  
(5.5)

To facilitate the subsequent computer analysis, dimensionless independent variables are defined as

\[
x' = x / L, \quad t' = t \alpha / L^2
\]  
(5.6)(5.7)

Substitution of Equations 5.6 and 5.7 in Equation 5.1 gives

\[
\frac{\partial u}{\partial (L^2 t'/\alpha)} = \alpha \frac{\partial^2 u}{\partial (L^2 x')^2}
\]

or

\[
\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x'^2}
\]  
(5.8)

where the prime (') has been dropped and it is understood that we are working with the dimensionless independent variables of Equations 5.6 and 5.7.

Equations 5.2 to 5.5 transform to (again, with the prime finally dropped).

\[
u(x, 0) = u_0
\]  
(5.9)

\[
-\frac{k}{L} \frac{\partial u(0, t)}{\partial x} = \sigma (a u_n^4 - e u(0, t))
\]  
(5.10)

\[
L_s \rho_s C_{ps} \frac{d u_s}{d (L^2 t'/\alpha)} = -\frac{k}{L} \frac{\partial u(1, t)}{\partial x'}
\]
or

\[
\frac{du_s}{dt} = -\left(\frac{kL}{\alpha}\right)\left(\frac{1}{Ls\rho_sC_{ps}}\right) \frac{\partial u(1, t)}{\partial x} \tag{5.11}
\]

\[
u_s(0) = u_0 \tag{5.12}
\]

Equations 5.8 to 5.12 are the model equations programmed in the six languages as discussed below. The parameters are taken as:

\[
k = 1.0 \text{ J} \cdot \text{m}^{-1} \cdot \text{s}^{-1} \cdot \text{K}^{-1}
\]

\[
\sigma = 5.67 \times 10^{-8} \text{ J} \cdot \text{m}^{-2} \cdot \text{K}^{-4}
\]

\[
\alpha = 1.0 \times 10^{-6} \text{m}^2/\text{s}
\]

\[
L = 0.1 \text{ m}
\]

\[
a = 1.0
\]

\[
e = 1.0
\]

\[
u_s = 2000 \text{ K}
\]

\[
u_0 = 25 \text{°C}
\]

\[
\rho_s = 7800.0 \text{ kg/m}^3
\]

\[
C_{ps} = 435.0 \text{ J/kg} \cdot \text{K}
\]

\[
L_s = 0.025 \text{ m}
\]

These values are typical for an insulation and steel system. Their particular significance is discussed subsequently when the numerical solution to Equations 5.8 to 5.12 is discussed.

The output is \(u(0, t), u(1/2, t), u(1, t)\) vs. \(t\) (in minutes; note \(t\) (minutes) = \(t\) (dimensionless) \(L^2/\alpha/60\)). In particular, we are interested in the steel temperature, \(u_s(t)\) as a function of \(t\), which is also the temperature of the insulation at the right face, \(u(1, t)\).

In constructing a program for Equations 5.8 to 5.12, we can make use of the methods of Chapter 4. In particular, we can approximate Equation 5.8 with the three point centered FD of Equation 4.9.

### 5.1 Programming in MATLAB

A main program for the solution of the nonlinear PDE problem, Equations 5.8 to 5.12, is the same as Program 3.1.1 and therefore is not listed here. `intpar`, `par1`, `par2`, `initial`, `derv`, and `fprint` are listed below followed by some discussion for each.

```matlab
function [neqn,nout,nsteps,t0,tf,abserr,relerr]=intpar

% Function intpar sets the parameters to control the
```
% integration of the nonlinear PDE problem
%
% Number of first order ODEs
neqn=22;
%
% Number of output points
nout=6;
%
% Maximum number of steps in the interval t0 to tf
nsteps=1000;
%
% Initial, final values of independent variable
% t0=0.0;
tf=0.2;
%
% Error tolerances
abserr=1.0e-01;
relerr=1.0e-05;

Program 5.1.1
intpar for the solution of Equations 5.8 to 5.12

We can note the following points about intpar:

- The number of ODEs is 22 (21 for Equation 5.8 and 1 for Equation 5.11):

  %
  % Number of first order ODEs
  neqn=22;

- 1000 steps produce a stable solution, which is apparently also accurate
  since the low-order (euler2b) and high-order (rkc4b, rkf45b) variable step
  ODE algorithms give essentially the same numerical solution (recall from
  Chapter 1 that in addition to varying the step size, i.e., h refinement,
  the comparison of solutions from algorithms of different orders, i.e.,
  p refinement, is a general way for assessing the accuracy of a numerical
  solution).

  %
  % Maximum number of steps in the interval t0 to tf
  nsteps=1000;

- The timescale for the problem is again 0 ≤ t ≤ 1 based on the dimensionless
  time of Equation 5.7:
% Initial, final values of independent variable
    t0=0.0;
    tf=0.2;

• The error tolerances are set to $abser = 10^{-1}$ (0.1 K) and $relerr = 10^{-5}$ (0.001%):

% Error tolerances
    abserr=1.0e-01;
    relerr=1.0e-05;

`inital` is listed below:

function [u]=inital(neqn,t)
% Function inital sets the initial condition vector for the nonlinear PDE
% Problem parameters
    [alpha,k,sigma,L,ui,ua]=par1;
% Initial condition
    for i=1:neqn
        u(i)=ui;
    end
% End of inital

Program 5.1.2
`inital` for the solution of Equations 5.8 to 5.12

`inital` is a straightforward implementation of Equations 5.9 and 5.12. Specifically, it calls function `par1` to set the initial temperature, $ui (= 25^\circ C = 298 K)$, then uses this initial value for the `neqn = 22` ODEs. `par1` and a second routine to set parameters, `par2`, are listed below (MATLAB would not accept a function with 12 output parameters, so two routines, `par1` and `par2`, are used, each with six parameters).

function [alpha,k,sigma,L,ui,ua]=par1
% Function par1 sets parameters for the nonlinear PDE
    alpha=1.0e-06;
    k=1.0;
\[
\begin{align*}
\sigma &= 5.67 \times 10^{-8} \\
L &= 0.1 \\
ui &= 298.0 \\
ua &= 2000.0 \\
\%
\%
\text{End of par1}
\end{align*}
\]

function \([a, e, \rho_S, \gamma, l_s, c_s] = \text{par2} \)
\%
\%
Function \text{par2} sets parameters for the nonlinear PDE
\%
a = 1.0; \\
e = 1.0; \\
\rho_S = 7800.0; \\
\gamma = 435.0; \\
l_s = 0.025; \\
c_s = \rho_S \gamma l_s;
\%
\%
End of par2

\textbf{Program 5.1.3}
par1 and par2 for the parameters of Equations 5.8 to 5.12

function \([u_t] = \text{derv}(\text{neqn}, t, u) \)
\%
\%
Function \text{derv} computes the derivative vector
\%
for the nonlinear PDE
\%
\%
Problem parameters
[alpha, k, sigma, L, ui, ua] = par1; \\
[ a, e, \rho_S, \gamma, l_s, c_s] = \text{par2};
\%
\%
Spatial grid
\[dx = 1.0/(\text{neqn}-2)\];
\[dx^2 = dx^2\];
\%
\%
Insulation
for \(i = 1: \text{neqn-2}\)
  if \(i = 1\)
    \[u_0 = u(2) + 2.0*dx*L*(\sigma/k)*(a*ua^4 - e*u(1)^4);\]
    \[u_t(1) = (u(2) - 2.0*u(1) + u_0)/dx^2;\]
  else
    \[u_t(i) = (u(i+1) - 2.0*u(i) + u(i-1))/dx^2;\]
  end
end
% Steel
    ut(neqn)=(k*L/alpha)*(1.0/cs)*(u(neqn-2)-u(neqn-1))/dx;
    ut(neqn-1)=ut(neqn);
%
% End of derv

Program 5.1.4
derv for the solution of Equations 5.8 to 5.12

We can note the following points about derv:

• The parameters in Equations 5.8 to 5.11 are first set by calls to par1 and par2:

    %
    % Problem parameters
    [alpha,k,sigma,L,ui,ua]=par1;
    [a,e, rhos,cps,ls,cs]=par2;

• Δx (= dx) is then computed. Note that the length of the system (the thickness of the insulation) is unity in terms of the dimensionless length of Equation 5.6:

    %
    % Spatial grid
    dx=1.0/(neqn-2);
    dxs=dx*dx;

• The method of lines solution of the PDE, Equation 5.8, is then programmed as a system of 20 ODEs (the 21st ODE is programmed subsequently by equating its derivative to that of the steel)

    %
    % Insulation
    for i=1:neqn-2
        if(i==1)
            u0=u(2)+2.0*dx*L*(sigma/k)*(a*ua^4-e*u(1)^4);
            ut(1)=(u(2)-2.0*u(1)+u0)/dxs;
        else
            ut(i)=(u(i+1)-2.0*u(i)+u(i-1))/dxs;
        end
    end
Note that for the first grid point \((i = 1)\), BC Equation 5.10, is approximated with a FD for the spatial derivative:

\[
-\frac{k}{L} \frac{\partial u(0, t)}{\partial x} \simeq -\frac{k}{L} \frac{u_2 - u_0}{2\Delta x} = \sigma (au_a^4 - eu_0^4) = \sigma (au_a^4 - eu_1^4)
\]

where \(u_0\) is a fictitious value that is outside the grid (the lowest grid index is \(i = 1\)). We can solve for \(u_0\)

\[
u_0 = u_2 + 2(L/k)\Delta x \sigma (au_a^4 - eu_1^4)
\]

(5.13)

then use this value for \(u_0\) in the FD approximation of Equation 5.8:

\[
\frac{du_1}{dt} \simeq \frac{u_2 - 2u_1 + u_0}{\Delta x^2}
\]

(5.14)

Equations 5.13 and 5.14 are programmed in the for loop for \(i = 1\).

- For \(i = 2\) to \(i = 20\), the FD approximation of Equation 4.9 can be used:

\[
\frac{du_1}{dt} = \frac{u_{i+1} - 2.0u_i + u_{i-1}}{\Delta x^2};
\]

- For ODEs 21 and 22, the programming is

\[
\% \% Steel
\]

\[
\frac{d}{dt}u(t) = \frac{u(\text{neqn})}{k^2/L^2} \cdot \text{cs} \cdot (u(t) - u(\text{neqn}));
\]

\[
u(\text{neqn}) = \text{ut}(\text{neqn}) - \text{ut}(\text{neqn} - 2)
\]

\[
u(\text{neqn}) = \text{ut}(\text{neqn} - 1)
\]

The programming for \(ut(\text{neqn})\) is Equation 5.11. Note that a FD approximation is used for the derivative \(\frac{\partial u(1, t)}{\partial x}\), i.e.,

\[
\frac{\partial u(1, t)}{\partial x} \simeq \frac{u_{21} - u_{20}}{\Delta x}
\]

The programming for \(ut(\text{neqn} - 1)\) merely sets \(u_{21}\) to \(u_{22}\) by equating their derivatives:

\[
\frac{\partial u(1, t)}{\partial t} = \frac{du_1}{dt}
\]

or

\[
\frac{du_{21}}{dt} = \frac{du_{22}}{dt}
\]

In other words, this provides continuity (equality) of the temperature between the right face of the insulation \((u(1, t))\) and the left face of the steel (with the uniform steel temperature \(u_s(t)\)).
At the end of *derv*, all 22 ODEs of Equations 5.8 and 5.11 are programmed. These 22 derivatives \( (ut(i)) \) are then returned to one of the ODE integrators, *euler2a* to *rkf45b*, to take the next step along the solution. In this way, the complete solution to Equations 5.8 to 5.12 is computed through \( t \) and \( x \); that is, we have \( u(x, t) \) and \( u_t(t) \) in numerical form.

This numerical solution is displayed by *fprint*:

function \[out\]=fprint(ncase,neqn,t,u)

% Function fprint displays the numerical solution to the nonlinear PDE
% Return current value of independent variable
% (MATLAB requires at least one return argument)
out=t;
%
% Problem parameters
[alpha,k,sigma,L,ui,ua]=par1;
[a,e,rhos(cps,ls,cs]=par2;
%
% Print a heading for the solution at t = 0
if(t<=0.0)

% Label for ODE integrator
% Fixed step modified Euler
if(ncase==1)
    fprintf('\n\n euler2a integrator\n');
%
% Variable step modified Euler
elseif(ncase==2)
    fprintf('\n\n euler2b integrator\n');
%
% Fixed step classical fourth order RK
elseif(ncase==3)
    fprintf('\n\n rkc4a integrator\n');
%
% Variable step classical fourth order RK
elseif(ncase==4)
    fprintf('\n\n rkc4b integrator\n');
%
% Fixed step RK Fehlberg 45
elseif(ncase==5)
    fprintf('\n\n rkf45a integrator\n');

Copyright © 2004 by Chapman & Hall/CRC
Variable step RK Fehlberg 45
elseif(ncase==6)
    fprintf('

rkf45b integrator
');
end

% Heading
fprintf('
ncase = %2d neqn = %2d
',ncase,neqn);
fprintf('
 t u(1) u(im) u(neqn)
);

% Grid index of midpoint
im=round(neqn)/2.0;

% Display the numerical solution
Tmin=t*L^2/alpha/60.0;
fprintf('%7.1f%10.2f%10.2f%10.2f
',tmin,u(1),u(im),u(neqn));

% End of fprintf

Program 5.1.5
fprintf for the solution of Equations 5.8 to 5.12

fprintf has the same function as the previous fprintf routines, i.e.,

- Labeling of the ODE integrator.
- Display of the numerical solution.

This completes the programming of Equations 5.8 to 5.12. The output is as follows:

euler2a integrator
ncase =  1  neqn = 22

<table>
<thead>
<tr>
<th>t</th>
<th>u(1)</th>
<th>u(im)</th>
<th>u(neqn)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>298.00</td>
<td>298.00</td>
<td>298.00</td>
</tr>
<tr>
<td>33.3</td>
<td>1987.96</td>
<td>1002.15</td>
<td>394.17</td>
</tr>
<tr>
<td>66.7</td>
<td>1991.05</td>
<td>1218.09</td>
<td>622.84</td>
</tr>
<tr>
<td>100.0</td>
<td>1992.55</td>
<td>1345.04</td>
<td>834.65</td>
</tr>
<tr>
<td>133.3</td>
<td>1993.72</td>
<td>1447.20</td>
<td>1015.31</td>
</tr>
<tr>
<td>166.7</td>
<td>1994.70</td>
<td>1533.03</td>
<td>1168.10</td>
</tr>
</tbody>
</table>
euler2b integrator

\[
\begin{array}{cccc}
\text{ncase} &=& 2 & \text{neqn} = 22 \\
\hline
\text{t} & u(1) & u(\text{im}) & u(\text{neqn}) \\
0.0 & 298.00 & 298.00 & 298.00 \\
33.3 & 1987.93 & 1002.11 & 394.14 \\
66.7 & 1991.00 & 1217.92 & 622.58 \\
100.0 & 1992.54 & 1344.68 & 834.02 \\
133.3 & 1993.68 & 1446.70 & 1014.43 \\
166.7 & 1994.65 & 1532.29 & 1166.78 \\
\end{array}
\]

rkc4a integrator

\[
\begin{array}{cccc}
\text{ncase} &=& 3 & \text{neqn} = 22 \\
\hline
\text{t} & u(1) & u(\text{im}) & u(\text{neqn}) \\
0.0 & 298.00 & 298.00 & 298.00 \\
33.3 & 1987.96 & 1002.16 & 394.17 \\
66.7 & 1991.05 & 1218.10 & 622.85 \\
100.0 & 1992.55 & 1345.05 & 834.66 \\
133.3 & 1993.72 & 1447.20 & 1015.32 \\
166.7 & 1994.70 & 1533.03 & 1168.10 \\
\end{array}
\]

rkc4b integrator

\[
\begin{array}{cccc}
\text{ncase} &=& 4 & \text{neqn} = 22 \\
\hline
\text{t} & u(1) & u(\text{im}) & u(\text{neqn}) \\
0.0 & 298.00 & 298.00 & 298.00 \\
33.3 & 1986.00 & 1002.16 & 394.17 \\
66.7 & 1991.04 & 1217.94 & 622.61 \\
100.0 & 1992.35 & 1344.91 & 834.42 \\
133.3 & 1993.56 & 1446.90 & 1014.78 \\
166.7 & 1994.69 & 1532.53 & 1167.22 \\
\end{array}
\]

rkf45a integrator

\[
\begin{array}{cccc}
\text{ncase} &=& 5 & \text{neqn} = 22 \\
\hline
\text{t} & u(1) & u(\text{im}) & u(\text{neqn}) \\
0.0 & 298.00 & 298.00 & 298.00 \\
33.3 & 1987.96 & 1002.16 & 394.17 \\
66.7 & 1991.05 & 1218.10 & 622.85 \\
100.0 & 1992.55 & 1345.05 & 834.66 \\
\end{array}
\]
We can note the following points about this output:

- The insulation starts at a uniform temperature of 298 K. The steel also starts at this temperature since, at the insulation–steel interface, \( u(1, t) = u_s(t) \) (from the coding in \( \text{derv} \)).
- As expected, the left face of the insulation (at \( x = 0 \)) responds rapidly to the flame and is above 1990 K for most of the transient.
- The insulation at the right face (\( x = L \)), which is also the steel temperature, responds more slowly due to the low thermal conductivity of the insulation. This delay in the rise of the steel temperature is critical in the case of structural steel used in buildings that may be subject to fire. This point is discussed further at the end of this chapter.

This completes the solution of Equations 5.8 to 5.12 with MATLAB. We now proceed to equivalent solutions in the five other languages.

### 5.2 Programming in C

Since main Program 2.2.1 is unchanged in the nonlinear PDE problem, it is not listed here. The header file, \texttt{intpar}, \texttt{par}, \texttt{initial}, \texttt{derv}, and \texttt{fprint} are listed below.

```c
/*
Definition of functions intpar, par, initial, derv, fprint
for the nonlinear PDE
*/
```
/* Include headers */
#include <math.h>
#include <stdio.h>

/* Maximum number of ODES */
#define SIZE 500

/* Type variables as extern (global) */
extern int neqn, nout, nsteps;
extern double t0, tf, abserr, relerr;
extern double alpha, k, sigma, L,
    ui, ua, ar, er,
    rhos, cps, ls, cs,
    xl, xu, dx, dxs;
extern FILE *fid;

/* Integration parameters */
void intpar();

/* Problem parameters */
void par();

/* Initial condition */
void initial(double u0[]);

/* Derivative vector */
void derv(double ut[], double t, double u[]);

/* Output */
void fprint(int ncase, double t, double u[]);

#include "pdenon.h"

#define SIZE 500

/* Type global variables */
int neqn, nout, nsteps;
double t0, tf, abserr, relerr;
double alpha, k, sigma, L,
    ui, ua, ar, er,
void intpar()

/* Function intpar sets the parameters to control the 
integration of the nonlinear PDE */
{
    /* Number of ODEs */
    neqn=22;

    /* Number of output points */
    nout=6;

    /* Maximum number of steps in the interval t0 to tf */
    nsteps=1000;

    /* Initial, final values of the independent variable */
    t0=0.0;
    tf=0.2;

    /* Error tolerances */
    abserr=pow(10.0,-1);
    relerr=pow(10.0,-5);

    /* End of intpar */
}

void par()

/* Function par sets the parameters for the nonlinear PDE */
{
    alpha=1.0e-06; k=1.0; sigma=5.67e-08; L=0.1;
    ui=298.0; ua=2000.0; ar=1.0; er=1.0;
    rhos=7800.0; cps=435.0; ls=0.025; cs=rhos*cps*ls;
    xl=0.0; xu=1.0; dx=(xu-xl)/(neqn-2); dxs=dx*dx;
}

void initial(double u0[])

Copyright 2004 by Chapman & Hall/CRC
/* Function initial sets the initial condition vector for the nonlinear PDE */
{
    /* Type variables */
    int i;

    /* Problem parameters */
    par();

    /* Initial condition */
    for(i=1;i<=neqn;i++)
    {
        u0[i]=ui;
    }

    /* End of initial */
}

void derv(double ut[], double t, double u[])
/* Function derv computes the derivative vector of the nonlinear PDE */
{
    /* Type variables */
    double u0;
    int i;

    /* Problem parameters */
    par();

    /* Insulation*/
    for(i=1;i<=(neqn-2);i++)
    {
        if(i==1)
        {
            u0=u[2]+2.0*dx*L*(sigma/k)*(ar*pow(ua,4)
            -er*pow(u[1],4));
            ut[1]=(u[2]-2.0*u[1]+u0)/dxs;
        }
        if(i>1)
        {
            ut[i]=(u[i+1]-2.0*u[i]+u[i-1])/dxs;
        }
    }
}
/* Steel */

ut[neqn] = (k*L/alpha)*(1.0/cs)*(u[neqn-2]-u[neqn-1])/dx;

ut[neqn-1] = ut[neqn];

/* End of derv */
}

void fprint(int ncase, double t, double u[])

/* Function fprint displays the numerical solution to the nonlinear PDE */
{
    /* Type variables */
    int im;

    /* Problem parameters */
    par();

    /* Print a heading for the solution at t = 0 */
    if(t<=0.0)
    {
        /* Label for ODE integrator */
        switch(ncase)
        {
            /*Fixed step modified Euler */
            case 1:
                fprintf(fid, "\n\n euler2a integrator\n\n");
                break;

            /* Variable step modified Euler */
            case 2:
                fprintf(fid, "\n\n euler2b integrator\n\n");
                break;

            /* Fixed step classical fourth order RK */
            case 3:
                fprintf(fid, "\n\n rkc4a integrator\n\n");
                break;

            /* Variable step classical fourth order RK */
            case 4:
                fprintf(fid, "\n\n rkc4b integrator\n\n");
                break;
        }
    }
}
 Program 5.2.1
Header file, intpar, par, initial, derv and fprint for the solution of Equations 5.8 to 5.12

These routines closely parallel the MATLAB routines in Section 5.1 and therefore will not be discussed here.

The output from the preceding Program 5.2.1 is listed below:

euler2a integrator

 ncase = 1  neqn = 22

t   u(1)   u(im)   u(neqn)
0    298.0   298.0    298.0
33   1988.0  1002.1   394.2
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>67</td>
<td>1991.0</td>
<td>1218.1</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>1992.6</td>
<td>1345.0</td>
</tr>
<tr>
<td></td>
<td>133</td>
<td>1993.7</td>
<td>1447.2</td>
</tr>
<tr>
<td></td>
<td>167</td>
<td>1994.7</td>
<td>1533.0</td>
</tr>
</tbody>
</table>

**euler2b integrator**

ncase = 2  neqn = 22

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>298.0</td>
<td>298.0</td>
</tr>
<tr>
<td></td>
<td>33</td>
<td>1987.9</td>
<td>1002.1</td>
</tr>
<tr>
<td></td>
<td>67</td>
<td>1991.0</td>
<td>1217.9</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>1992.5</td>
<td>1344.7</td>
</tr>
<tr>
<td></td>
<td>133</td>
<td>1993.7</td>
<td>1446.7</td>
</tr>
<tr>
<td></td>
<td>167</td>
<td>1994.7</td>
<td>1532.3</td>
</tr>
</tbody>
</table>

**rkc4a integrator**

ncase = 3  neqn = 22

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>298.0</td>
<td>298.0</td>
</tr>
<tr>
<td></td>
<td>33</td>
<td>1988.0</td>
<td>1002.2</td>
</tr>
<tr>
<td></td>
<td>67</td>
<td>1991.0</td>
<td>1218.1</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>1992.6</td>
<td>1345.0</td>
</tr>
<tr>
<td></td>
<td>133</td>
<td>1993.7</td>
<td>1447.2</td>
</tr>
<tr>
<td></td>
<td>167</td>
<td>1994.7</td>
<td>1533.0</td>
</tr>
</tbody>
</table>

**rkc4b integrator**

ncase = 4  neqn = 22

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>298.0</td>
<td>298.0</td>
</tr>
<tr>
<td></td>
<td>33</td>
<td>1986.0</td>
<td>1002.2</td>
</tr>
<tr>
<td></td>
<td>67</td>
<td>1991.0</td>
<td>1217.9</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>1992.3</td>
<td>1344.9</td>
</tr>
<tr>
<td></td>
<td>133</td>
<td>1993.6</td>
<td>1446.9</td>
</tr>
<tr>
<td></td>
<td>167</td>
<td>1994.7</td>
<td>1532.5</td>
</tr>
</tbody>
</table>
rkf45a integrator
ncase = 5  neqn = 22

<table>
<thead>
<tr>
<th>t</th>
<th>u(1)</th>
<th>u(im)</th>
<th>u(neqn)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>298.0</td>
<td>298.0</td>
<td>298.0</td>
</tr>
<tr>
<td>33</td>
<td>1988.0</td>
<td>1002.2</td>
<td>394.2</td>
</tr>
<tr>
<td>67</td>
<td>1991.0</td>
<td>1218.1</td>
<td>622.8</td>
</tr>
<tr>
<td>100</td>
<td>1992.6</td>
<td>1345.0</td>
<td>834.7</td>
</tr>
<tr>
<td>133</td>
<td>1993.7</td>
<td>1447.2</td>
<td>1015.3</td>
</tr>
<tr>
<td>167</td>
<td>1994.7</td>
<td>1533.0</td>
<td>1168.1</td>
</tr>
</tbody>
</table>

rkf45b integrator
ncase = 6  neqn = 22

<table>
<thead>
<tr>
<th>t</th>
<th>u(1)</th>
<th>u(im)</th>
<th>u(neqn)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>298.0</td>
<td>298.0</td>
<td>298.0</td>
</tr>
<tr>
<td>33</td>
<td>1988.0</td>
<td>1002.2</td>
<td>394.2</td>
</tr>
<tr>
<td>67</td>
<td>1991.3</td>
<td>1218.1</td>
<td>622.8</td>
</tr>
<tr>
<td>100</td>
<td>1992.5</td>
<td>1345.0</td>
<td>834.7</td>
</tr>
<tr>
<td>133</td>
<td>1993.9</td>
<td>1447.0</td>
<td>1015.0</td>
</tr>
<tr>
<td>167</td>
<td>1994.7</td>
<td>1532.6</td>
<td>1167.4</td>
</tr>
</tbody>
</table>

This output is essentially the same as the MATLAB output in Section 5.1. We now proceed to the C++ programming of the nonlinear PDE.

5.3 Programming in C++

Again, since main Program 2.3.1 is unchanged, it is not listed here. The associated header files are listed below:

```c
/* Problem parameters */
void par();

/* Initial condition */
void inital();

/* Derivative vector */
void derv(double ut[], double t, double u[]);

/* Output */
void fprint(int ncase, int neqn, double t, double u[]);
```

Copyright 2004 by Chapman & Hall/CRC


Program 5.3.1
Header files pdenon.h and MOL.h for the solution of Equations 5.8 to 5.12

\texttt{intpar}, \texttt{par}, \texttt{inital}, \texttt{derv}, and \texttt{fprint} are listed below:

\begin{verbatim}
#include "DEF.h"

void DEF::intpar()
{
/* Function intpar sets the parameters to control the integration of the nonlinear PDE */
{
    /* Number of ODEs */
    neqn=22;

    /* Number of output points */
    nout=6;

    /* Maximum number of steps in the interval t0 to tf */
    nsteps=1000;

    /* Initial, final values of the independent variable */
    t0=0.0;
    tf=0.2;
}
\end{verbatim}
/* Error tolerances */
abserr=pow(10.0,-1);
relerr=pow(10.0,-5);

/* End of intpar */
}

void DEF::par()
/* Function par sets the parameters for the nonlinear PDE */
{
  alpha=1.0e-06;  k=1.0;  sigma=5.67e-08;  L=0.1;
  ui=298.0;  ua=2000.0;  ar=1.0;  er=1.0;
  rhos=7800.0;  cps=435.0;  ls=0.025;  cs=rhos*cps*ls;
  xl=0.0;  xu=1.0;  dx=(xu-xl)/(neqn-2);  dxs=dx*dx;

  /* End of par */
}

void DEF::inital()
/* Function inital sets the initial condition vector for
the nonlinear PDE */
{
  /* Type variables */
  int i;

  /* Initial condition */
  for(i=1;i<=neqn;i++)
    { u0[i]=ui; }

  /* End of inital */
}

void DEF::derv(double ut[], double t, double u[])
/* Function derv computes the derivative vector of the
nonlinear PDE problem */
{
  /* Type variables */
  double u0;
  int i;
/* Integration parameters */
intpar();

/* Problem parameters */
par();

/* Insulation */
for(i=1;i<=(neqn-2);i++)
{
  if(i==1)
  {
    u0=u[2]+2.0*dx*L*(sigma/k)*(ar*pow(ua,4)
    -er*pow(u[1],4));
    ut[1]=(u[2]-2.0*u[1]+u0)/dxs;
  }
  if(i>1)
  {
    ut[i]=(u[i+1]-2.0*u[i]+u[i-1])/dxs;
  }
}
/* Steel */
ut[neqn]=(k*L/alpha)*(1.0/cs)*(u[neqn-2]-u[neqn-1])/dx;
ut[neqn-1]=ut[neqn];

/* End of derv */
}

void DEF::fprint(int ncase, int neqn, double t, double u[])

/* Function fprint displays the numerical solution to the
 nonlinear PDE */
{
  /* Type variables */
  int im;

  /* Problem parameters */
  par();

  /* Print a heading for the solution at t = 0 */
  if(t<=0.0)
  {
    /* Label for ODE integrator */
    switch(ncase)
{ /* Fixed step modified Euler */
case 1:
fprintf(fid,"\n\n euler2a integrator\n");
break;

/* Variable step modified Euler */
case 2:
fprintf(fid,"\n\n euler2b integrator\n");
break;

/* Fixed step classical fourth order RK */
case 3:
fprintf(fid,"\n\n rkc4a integrator\n");
break;

/* Variable step classical fourth order RK */
case 4:
fprintf(fid,"\n\n rkc4b integrator\n");
break;

/* Fixed step RK Fehlberg 45 */
case 5:
fprintf(fid,"\n\n rkf45a integrator\n");
break;

/* Variable step RK Fehlberg 45 */
case 6:
fprintf(fid,"\n\n rkf45b integrator\n");
break;
}

/* Heading */
fprintf(fid,"\n ncase = %1d, neqn = %2d\n\n",ncase,neqn);
fprintf(fid," t  u(1)  u(im)  u(neqn)\n");

/* End of t = 0 heading */
}

/* Grid index of midpoint */
im=(neqn+1)/2;

/* Display the numerical solution */
Program 5.3.2
intpar, par, initial, derv, and fprintf for the solution of Equations 5.8 to 5.12

The output from the routines of Programs 5.3.1 and 5.3.2 is listed below:

euler2a integrator
ncase = 1, neqn = 22

<table>
<thead>
<tr>
<th>t</th>
<th>u(1)</th>
<th>u(im)</th>
<th>u(neqn)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>298.0</td>
<td>298.0</td>
<td>298.0</td>
</tr>
<tr>
<td>33</td>
<td>1988.0</td>
<td>1002.1</td>
<td>394.2</td>
</tr>
<tr>
<td>67</td>
<td>1991.0</td>
<td>1218.1</td>
<td>622.8</td>
</tr>
<tr>
<td>100</td>
<td>1992.6</td>
<td>1345.0</td>
<td>834.7</td>
</tr>
<tr>
<td>133</td>
<td>1993.7</td>
<td>1447.2</td>
<td>1015.3</td>
</tr>
<tr>
<td>167</td>
<td>1994.7</td>
<td>1533.0</td>
<td>1168.1</td>
</tr>
</tbody>
</table>

euler2b integrator
ncase = 2, neqn = 22

<table>
<thead>
<tr>
<th>t</th>
<th>u(1)</th>
<th>u(im)</th>
<th>u(neqn)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>298.0</td>
<td>298.0</td>
<td>298.0</td>
</tr>
<tr>
<td>33</td>
<td>1987.9</td>
<td>1002.1</td>
<td>394.1</td>
</tr>
<tr>
<td>67</td>
<td>1991.0</td>
<td>1217.9</td>
<td>622.6</td>
</tr>
<tr>
<td>100</td>
<td>1992.5</td>
<td>1344.7</td>
<td>834.0</td>
</tr>
<tr>
<td>133</td>
<td>1993.7</td>
<td>1446.7</td>
<td>1014.4</td>
</tr>
<tr>
<td>167</td>
<td>1994.7</td>
<td>1532.3</td>
<td>1166.8</td>
</tr>
</tbody>
</table>

rkc4a integrator
ncase = 3, neqn = 22

<table>
<thead>
<tr>
<th>t</th>
<th>u(1)</th>
<th>u(im)</th>
<th>u(neqn)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>298.0</td>
<td>298.0</td>
<td>298.0</td>
</tr>
<tr>
<td>33</td>
<td>1988.0</td>
<td>1002.2</td>
<td>394.2</td>
</tr>
<tr>
<td>67</td>
<td>1991.0</td>
<td>1218.1</td>
<td>622.8</td>
</tr>
<tr>
<td>100</td>
<td>1992.6</td>
<td>1345.0</td>
<td>834.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>133</td>
<td>1993.7</td>
<td>1447.2</td>
<td>1015.3</td>
</tr>
<tr>
<td>167</td>
<td>1994.7</td>
<td>1533.0</td>
<td>1168.1</td>
</tr>
</tbody>
</table>

**rkc4b integrator**

ncase = 4, neqn = 22

<table>
<thead>
<tr>
<th>t</th>
<th>u(1)</th>
<th>u(im)</th>
<th>u(neqn)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>298.0</td>
<td>298.0</td>
<td>298.0</td>
</tr>
<tr>
<td>33</td>
<td>1986.0</td>
<td>1002.2</td>
<td>394.2</td>
</tr>
<tr>
<td>67</td>
<td>1991.0</td>
<td>1217.9</td>
<td>622.6</td>
</tr>
<tr>
<td>100</td>
<td>1992.3</td>
<td>1344.9</td>
<td>834.4</td>
</tr>
<tr>
<td>133</td>
<td>1993.6</td>
<td>1446.9</td>
<td>1014.8</td>
</tr>
<tr>
<td>167</td>
<td>1994.7</td>
<td>1532.5</td>
<td>1167.2</td>
</tr>
</tbody>
</table>

**rkf45a integrator**

ncase = 5, neqn = 22

<table>
<thead>
<tr>
<th>t</th>
<th>u(1)</th>
<th>u(im)</th>
<th>u(neqn)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>298.0</td>
<td>298.0</td>
<td>298.0</td>
</tr>
<tr>
<td>33</td>
<td>1988.0</td>
<td>1002.2</td>
<td>394.2</td>
</tr>
<tr>
<td>67</td>
<td>1991.0</td>
<td>1218.1</td>
<td>622.8</td>
</tr>
<tr>
<td>100</td>
<td>1992.6</td>
<td>1345.0</td>
<td>834.7</td>
</tr>
<tr>
<td>133</td>
<td>1993.7</td>
<td>1447.2</td>
<td>1015.3</td>
</tr>
<tr>
<td>167</td>
<td>1994.7</td>
<td>1533.0</td>
<td>1168.1</td>
</tr>
</tbody>
</table>

**rkf45b integrator**

ncase = 6, neqn = 22

<table>
<thead>
<tr>
<th>t</th>
<th>u(1)</th>
<th>u(im)</th>
<th>u(neqn)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>298.0</td>
<td>298.0</td>
<td>298.0</td>
</tr>
<tr>
<td>33</td>
<td>1988.0</td>
<td>1002.2</td>
<td>394.2</td>
</tr>
<tr>
<td>67</td>
<td>1990.0</td>
<td>1218.1</td>
<td>622.8</td>
</tr>
<tr>
<td>100</td>
<td>1989.5</td>
<td>1345.0</td>
<td>834.6</td>
</tr>
<tr>
<td>133</td>
<td>1993.7</td>
<td>1446.9</td>
<td>1014.8</td>
</tr>
<tr>
<td>167</td>
<td>1994.6</td>
<td>1532.7</td>
<td>1167.5</td>
</tr>
</tbody>
</table>

This output is equivalent to the output from the preceding MATLAB (Section 5.1) and C (Section 5.2). We now proceed to the Fortran programming of the nonlinear PDE.
5.4 Programming in Fortran

Again, since main Program 2.4.1 is unchanged, it is not listed here. intpar, par, initial, dero, and fprint are listed below:

```fortran
subroutine intpar(neqn,nout,nsteps,t0,tf,abserr,relerr)
  C
  C Subroutine intpar sets the parameters to control the
  C integration of the nonlinear PDE problem
  C
  C Double precision coding is used
    implicit double precision(a-h,o-z)
  C
  C Number of ODEs
    neqn=22
  C
  C Number of output points
    nout=6
  C
  C Maximum number of steps in the interval t0 to tf
    nsteps=1000
  C
  C Initial, final values of the independent variable
    t0=0.0d0
    tf=0.2d0
  C
  C Error tolerances
    abserr=1.0d-01
    relerr=1.0d-05
    return
  C
  C End of intpar
  end

subroutine par
  C
  C Double precision coding is used
    implicit double precision(a-h,o-z)
  C
  C Global variables
    common/p/ alpha, xk, sigma, xl, ui, ua,
    + a, e, rhos, cps, xls, cs
```
C Subroutine par sets the parameters for the nonlinear PDE
   alpha=1.0d-06
   xk=1.0d0
   sigma=5.67d-08
   xl=0.1d0
   ui=298.0d0
   ua=2000.0d0
   a=1.0d0
   e=1.0d0
   rhos=7800.0d0
   cps=435.0d0
   xls=0.025d0
   cs=rhos*cps*xls
return
C End of par
end

subroutine initial(neqn,t,u0)
C Subroutine initial sets the initial condition vector
C for the nonlinear PDE problem
C
C Double precision coding is used
   implicit double precision(a-h,o-z)
C
C Global variables
   common/p/ alpha, xk, sigma, xl, ui, ua,
   a, e, rhos, cps, xls, cs
C
C Size the arrays
   dimension u0(neqn)
C
C Problem parameters
   call par
C
C Initial condition
   do i=1,neqn
      u0(i)=ui
   end do
   return
C End of initial
end
subroutine derv(neqn,t,u,ut)
C
C Subroutine derv computes the derivative vector
C of the nonlinear PDE problem
C
C Double precision coding is used
    implicit double precision(a-h,o-z)
C
C Global variables
    common/p/ alpha, xk, sigma, xl, ui, ua,
    + a, e, rhos, cps, xls, cs
C
C Size the arrays
    dimension u(neqn), ut(neqn)
C
C Spatial grid
    dx=1.0d0/dfloat(neqn-2)
    dxs=dx*dx
C
C Insulation
    do i=1,neqn-2
        if(i.eq.1)then
            u0=u(2)+2.0d0*dx*xl*(sigma/xk)*(a*ua**4-e*u(1)**4)
            ut(i)=(u(2)-2.0d0*u(1)+u0)/dxs
        else
            ut(i)=(u(i+1)-2.0d0*u(i)+u(i-1))/dxs
        end if
    end do
C
C Steel
    ut(neqn)=(xk*xl/alpha)*(1.0d0/cs)*(u(neqn-2)
    -u(neqn-1))/dx
    ut(neqn-1)=ut(neqn)
C
C End of derv
    return
end

subroutine fprint(no,ncase,neqn,t,u)
C
C Subroutine fprint displays the numerical solution to the
C nonlinear PDE problem
C
C Double precision coding is used
    implicit double precision(a-h,o-z)
C
C Global variables
    common/p/ alpha, xk, sigma, xl, ui, ua,
    + a, e, rhos, cps, xls, cs
C
C Size the arrays
    dimension u(neqn)
C
C Print a heading for the solution at t = 0
    if(t.le.0.0d0)then
    C
C Label for ODE integrator
    C
C Fixed step modified Euler
    if(ncase.eq.1)then
        write(no,11)
        11 format(//, ' euler2a integrator')
    C
C Variable step modified Euler
    else if(ncase.eq.2)then
        write(no,12)
        12 format(//, ' euler2b integrator')
    C
C Fixed step classical fourth order RK
    else if(ncase.eq.3)then
        write(no,13)
        13 format(//, ' rkc4a integrator')
    C
C Variable step classical fourth order RK
    else if(ncase.eq.4)then
        write(no,14)
        14 format(//, ' rkc4b integrator')
    C
C Fixed step RK Fehlberg 45
    else if(ncase.eq.5)then
        write(no,15)
        15 format(//, ' rkf45a integrator')
    C
C Variable step RK Fehlberg 45
    else if(ncase.eq.6)then
        write(no,16)
        16 format(//, ' rkf45b integrator')
    end if
C
C Heading
Program 5.4.1

intpar, par, initial, deriv, and fprint for the solution of Equations 5.8 to 5.12

The output from the routines of Program 5.4.1 is listed below:

euler2a integrator

ncase =  1  neqn = 22

t  u(1)  u(im)  u(neqn)
    0.0  298.00  298.00  298.00
   33.3 1997.96 1002.15  394.17
   66.7 1991.05 1218.09  622.84
  100.0 1992.55 1345.04  834.65
  133.3 1993.72 1447.20 1015.31
  166.7 1994.70 1533.03 1168.10

euler2b integrator

ncase =  2  neqn = 22

t  u(1)  u(im)  u(neqn)
    0.0  298.00  298.00  298.00
rkc4a integrator
ncase = 3 neqn = 22

<table>
<thead>
<tr>
<th>t</th>
<th>u(1)</th>
<th>u(im)</th>
<th>u(neqn)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>298.00</td>
<td>298.00</td>
<td>298.00</td>
</tr>
<tr>
<td>33.3</td>
<td>1987.96</td>
<td>1002.11</td>
<td>394.14</td>
</tr>
<tr>
<td>66.7</td>
<td>1991.00</td>
<td>1217.92</td>
<td>622.58</td>
</tr>
<tr>
<td>100.0</td>
<td>1992.54</td>
<td>1344.68</td>
<td>834.02</td>
</tr>
<tr>
<td>133.3</td>
<td>1993.68</td>
<td>1446.70</td>
<td>1014.43</td>
</tr>
<tr>
<td>166.7</td>
<td>1994.65</td>
<td>1532.29</td>
<td>1166.78</td>
</tr>
</tbody>
</table>

rkc4b integrator
ncase = 4 neqn = 22

<table>
<thead>
<tr>
<th>t</th>
<th>u(1)</th>
<th>u(im)</th>
<th>u(neqn)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>298.00</td>
<td>298.00</td>
<td>298.00</td>
</tr>
<tr>
<td>33.3</td>
<td>1987.96</td>
<td>1002.16</td>
<td>394.17</td>
</tr>
<tr>
<td>66.7</td>
<td>1991.05</td>
<td>1218.10</td>
<td>622.85</td>
</tr>
<tr>
<td>100.0</td>
<td>1992.55</td>
<td>1345.05</td>
<td>834.66</td>
</tr>
<tr>
<td>133.3</td>
<td>1993.72</td>
<td>1447.20</td>
<td>1015.32</td>
</tr>
<tr>
<td>166.7</td>
<td>1994.70</td>
<td>1533.03</td>
<td>1168.10</td>
</tr>
</tbody>
</table>

rkf45a integrator
ncase = 5 neqn = 22

<table>
<thead>
<tr>
<th>t</th>
<th>u(1)</th>
<th>u(im)</th>
<th>u(neqn)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>298.00</td>
<td>298.00</td>
<td>298.00</td>
</tr>
<tr>
<td>33.3</td>
<td>1987.96</td>
<td>1002.16</td>
<td>394.17</td>
</tr>
<tr>
<td>66.7</td>
<td>1991.05</td>
<td>1218.10</td>
<td>622.85</td>
</tr>
<tr>
<td>100.0</td>
<td>1992.55</td>
<td>1345.05</td>
<td>834.66</td>
</tr>
<tr>
<td>133.3</td>
<td>1993.72</td>
<td>1447.20</td>
<td>1015.32</td>
</tr>
<tr>
<td>166.7</td>
<td>1994.70</td>
<td>1533.03</td>
<td>1168.10</td>
</tr>
</tbody>
</table>
This output is equivalent to the output from the preceding MATLAB (Section 5.1), C (Section 5.2), and C++ (Section 5.3). We now proceed to the Java programming of the nonlinear PDE.

5.5 Programming in Java

Again, since main Program 2.5.1 is unchanged in the nonlinear PDE problem, it is not listed here. An interface routine is listed below:

```java
package mol;

public class MOL
{
    /* Maximum (default) number of ODEs */
    public int SIZE=500;

    /* Variables for ODE integration */
    public int neqn, nout, nsteps;
    public double t0, tf, abserr, relerr;
    public double u[], u0[], e[];
```
/* Parameters for PDE */
    public double
        alpha, k, sigma, L,
        ui, ua, ar, er,
        rhos, cps, ls, cs=rhos*cps*ls,
        xl, xu, dx=(xu-xl)/(neqn-2), dxs=dx*dx;
    public MOL()
    {
    }
}

Program 5.5.1
Interface routine for the solution of Equations 5.8 to 5.12

    intpar, par, initial, deriv, and fprint are listed next:

    /* This file is a member of the package mol */
    package mol;

    import mol.MOL;
    import java.math.*;
    import java.io.*;
    import java.text.*;

    public class DEF extends MOL implements pdenoninterface
    {

        public DEF()
        {
            /* Integration parameters */
            this.intpar();

            /* Declare arrays */
            u0=new double[SIZE];
            u=new double[SIZE];
            e=new double[SIZE];

            /* Problem parameters */
            this.par();

            /* Initial condition vector */
            this.initial();

        }
    }
public void intpar()
{
    /* Function intpar sets the parameters to control the integration of the nonlinear PDE */
    /* Number of ODEs */
    neqn=22;
    
    /* Size of arrays in MOL library */
    SIZE=neqn+1;
    
    /* Number of output points */
    nout=6;
    
    /* Maximum number of steps in the interval t0 to tf */
    nsteps=1000;
    
    /* Initial, final values of the independent variable */
    t0=0.0;
    tf=0.2;
    
    /* Error tolerances */
    abserr=Math.pow(10.0,-1.0);
    relerr=Math.pow(10.0,-5.0);
    
    /* End of intpar */
}

public void par()
{
    /* Function par sets the parameters for the nonlinear PDE */
    alpha=1.0e-06; k=1.0; sigma=5.67e-08; L=0.1;
    ui=298.0; ua=2000.0; ar=1.0; er=1.0;
    rhos=7800.0; cps=435.0; ls=0.025; cs=rhos*cps*ls;
    x1=0.0; xu=1.0; dx=(xu-x1)/(neqn-2); dxs=dx*dx;
    
    /* End of par */
}

public void initial()
{
    /* Function initial sets the initial condition vector for the nonlinear PDE */

{ /* Type variables */
  int i;

  /* Initial condition */
  for(i=1;i<=neqn;i++)
    { u0[i]=ui; }

  /* End of initial */
}

public void derv(double ut[], double t, double u[])

  /* Function derv computes the derivative vector of the nonlinear PDE */
  {
    /* Type variables */
    double u0;
    int i;

    /* Problem parameters */
    par();

    /* Insulation */
    for(i=1;i<=(neqn-1);i++)
      { if(i==1) 
          { u0=u[2]+2.0*dx*L*(sigma/k)*(ar*Math.pow(ua,4) -er*Math.pow(u[1],4)); ut[1]=(u[2]-2.0*u[1]+u0)/dxs; } 
        if(i>1) 
          { ut[i]=(u[i+1]-2.0*u[i]+u[i-1])/dxs; } 
      } 

    /* Steel */
    ut[neqn]=(k*L/alpha)*(1.0/cs)*(u[neqn-2]-u[neqn-1])/dx; 
    ut[neqn-1]=ut[neqn];
public void fprint(PrintWriter f, int ncase, int neqn, double t, double u[])

/* Function fprint displays the numerical solution to the nonlinear PDE */
{
    /* Type variables */
    int im;

    /* Print a heading for the solution at t = 0 */
    if(t<=0.0)
    {
        /* Label for ODE integrator */
        switch(ncase)
        {
            /*Fixed step modified Euler */
            case 1:
                f.println("\n\n euler2a integrator\n");
                break;

            /* Variable step modified Euler */
            case 2:
                f.println("\n\n euler2b integrator\n");
                break;

            /* Fixed step classical fourth order RK */
            case 3:
                f.println("\n\n rkc4a integrator\n");
                break;

            /* Variable step classical fourth order RK */
            case 4:
                f.println("\n\n rkc4b integrator\n");
                break;

            /* Fixed step RK Fehlberg 45 */
            case 5:
                f.println("\n\n rkf45a integrator\n");
                break;

            /* Variable step RK Fehlberg 45 */
        }
    }
}
case 6:
    f.println("\n\n rkf45b integrator\n");
    break;
}

/* Heading */
f.println(" neqn = "+neqn+ " ncase = "+ncase+ " \n");
f.println(" t u(1) u(im) u(neqn)\n");

/* End of t = 0 heading */
}

/* Grid index of midpoint */
im=(neqn+1)/2;

/* Display format for floating numbers */
DecimalFormat df1=new DecimalFormat("0.00");
DecimalFormat df2=new DecimalFormat("0.0");

/* Display the numerical solution */
f.println(df1.format(t) +"\t " +df2.format(u[1]) +"\t " +df2.format(u[im]) +"\t " +df2.format(u[neqn]));

/* End of fprintf */
}

/* End of DEF */
}

Program 5.5.2
intpar, par, initial, derv, and fprintf for the solution of Equations 5.8 to 5.12

The output from the preceding routines is listed below:

euler2a integrator

neqn = 22 ncase = 1

t   u(1)   u(im)   u(neqn)
0.00 298.0 298.0 298.0
0.20 1988.0 1002.1 394.2
0.40 1991.0 1218.1 622.8

Copyright 2004 by Chapman & Hall/CRC
<table>
<thead>
<tr>
<th>t</th>
<th>u(1)</th>
<th>u(im)</th>
<th>u(neqn)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>298.0</td>
<td>298.0</td>
<td>298.0</td>
</tr>
<tr>
<td>0.20</td>
<td>1987.9</td>
<td>1002.1</td>
<td>394.1</td>
</tr>
<tr>
<td>0.40</td>
<td>1991.0</td>
<td>1217.9</td>
<td>622.6</td>
</tr>
<tr>
<td>0.60</td>
<td>1992.5</td>
<td>1344.7</td>
<td>834.0</td>
</tr>
<tr>
<td>0.80</td>
<td>1993.7</td>
<td>1446.7</td>
<td>1014.4</td>
</tr>
<tr>
<td>1.00</td>
<td>1994.7</td>
<td>1532.3</td>
<td>1166.8</td>
</tr>
</tbody>
</table>

**rkc4a integrator**

<table>
<thead>
<tr>
<th>t</th>
<th>u(1)</th>
<th>u(im)</th>
<th>u(neqn)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>298.0</td>
<td>298.0</td>
<td>298.0</td>
</tr>
<tr>
<td>0.20</td>
<td>1988.0</td>
<td>1002.2</td>
<td>394.2</td>
</tr>
<tr>
<td>0.40</td>
<td>1991.0</td>
<td>1218.1</td>
<td>622.8</td>
</tr>
<tr>
<td>0.60</td>
<td>1992.6</td>
<td>1345.0</td>
<td>834.7</td>
</tr>
<tr>
<td>0.80</td>
<td>1993.7</td>
<td>1447.2</td>
<td>1015.3</td>
</tr>
<tr>
<td>1.00</td>
<td>1994.7</td>
<td>1533.0</td>
<td>1168.1</td>
</tr>
</tbody>
</table>

**rkc4b integrator**

<table>
<thead>
<tr>
<th>t</th>
<th>u(1)</th>
<th>u(im)</th>
<th>u(neqn)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>298.0</td>
<td>298.0</td>
<td>298.0</td>
</tr>
<tr>
<td>0.20</td>
<td>1986.0</td>
<td>1002.2</td>
<td>394.2</td>
</tr>
<tr>
<td>0.40</td>
<td>1991.0</td>
<td>1217.9</td>
<td>622.6</td>
</tr>
<tr>
<td>0.60</td>
<td>1992.3</td>
<td>1344.9</td>
<td>834.4</td>
</tr>
<tr>
<td>0.80</td>
<td>1993.6</td>
<td>1446.9</td>
<td>1014.8</td>
</tr>
<tr>
<td>1.00</td>
<td>1994.7</td>
<td>1532.5</td>
<td>1167.2</td>
</tr>
</tbody>
</table>
rkf45a integrator

neqn = 22  ncase = 5

<table>
<thead>
<tr>
<th>t</th>
<th>u(1)</th>
<th>u(im)</th>
<th>u(neqn)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>298.0</td>
<td>298.0</td>
<td>298.0</td>
</tr>
<tr>
<td>0.20</td>
<td>1988.0</td>
<td>1002.2</td>
<td>394.2</td>
</tr>
<tr>
<td>0.40</td>
<td>1991.0</td>
<td>1218.1</td>
<td>622.8</td>
</tr>
<tr>
<td>0.60</td>
<td>1992.6</td>
<td>1345.0</td>
<td>834.7</td>
</tr>
<tr>
<td>0.80</td>
<td>1993.7</td>
<td>1447.2</td>
<td>1015.3</td>
</tr>
<tr>
<td>1.00</td>
<td>1994.7</td>
<td>1533.0</td>
<td>1168.1</td>
</tr>
</tbody>
</table>

rkf45b integrator

neqn = 22  ncase = 6

<table>
<thead>
<tr>
<th>t</th>
<th>u(1)</th>
<th>u(im)</th>
<th>u(neqn)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>298.0</td>
<td>298.0</td>
<td>298.0</td>
</tr>
<tr>
<td>0.20</td>
<td>1988.0</td>
<td>1002.2</td>
<td>394.2</td>
</tr>
<tr>
<td>0.40</td>
<td>1991.3</td>
<td>1218.1</td>
<td>622.8</td>
</tr>
<tr>
<td>0.60</td>
<td>1992.5</td>
<td>1345.0</td>
<td>834.7</td>
</tr>
<tr>
<td>0.80</td>
<td>1993.9</td>
<td>1447.0</td>
<td>1015.0</td>
</tr>
<tr>
<td>1.00</td>
<td>1994.7</td>
<td>1532.6</td>
<td>1167.4</td>
</tr>
</tbody>
</table>

This output is equivalent to the output from the preceding MATLAB (Section 5.1), C (Section 5.2), C++ (Section 5.3), and Fortran (Section 5.4). We now proceed to the Maple programming of the nonlinear PDE.

5.6 Programming in Maple

Since main Program 5.6.1 (and subordinate routines) accesses specific files by read statements, it is listed first:

```plaintext
> restart:

> read "c:\odelib\maple\pdenon\pdenon.txt";

> pdenon();
```

Program 5.6.1
Maple main program pdenon.mws for the numerical integration of Equations 5.8 to 5.12
pdenon:=proc() 
# 
# Main program pdenon computes the numerical 
# solution to the nonlinear PDE system by one of 
# six integrators 
# 
# Type variables 
    global neqn, nout, nsteps, t0, tf, abserr, relerr: 
    local u0, u, tp, ncase, i, j: 
# 
# Step through six integrators 
    for ncase from 1 to 6 do 
# 
# Integration parameters 
        read "c:\odelib\maple\pdenon\intpar.txt": 
        intpar(): 
# 
# Size arrays 
        u0:=array(1..neqn): u:=array(1..neqn): 
# 
# Initial condition vector 
        read "c:\odelib\maple\pdenon\inital.txt": 
        inital(neqn,t0,u0): 
# 
# Output interval 
        tp:=tf-t0: 
# 
# Compute solution at nout output points 
    for j from 1 to nout do 
# 
# Print current solution 
        read "c:\odelib\maple\pdenon\fprint.txt": 
        fprint(ncase,neqn,t0,u0): 
# 
# Fixed step modified Euler integrator 
    if (ncase = 1) then 
        read "c:\odelib\maple\pdenon\euler2a.txt": 
        euler2a(neqn,t0,tf,u0,nsteps,u): 
        end if: 
# 
# Variable step modified Euler integrator 
    if (ncase = 2) then 
        read "c:\odelib\maple\pdenon\euler2b.txt": 
        euler2b(neqn,t0,tf,u0,nsteps,abserr,relerr,u): 
        end if: 

# Fixed step classical fourth order RK integrator
if (ncase = 3) then
    read "c:\odelib\maple\pdenon\rkc4a.txt":
    rkc4a(neqn,t0,tf,u0,nsteps,u):
end if:
#
# Variable step classical fourth order RK integrator
if (ncase = 4) then
    read "c:\odelib\maple\pdenon\rkc4b.txt":
    rkc4b(neqn,t0,tf,u0,nsteps,abserr,relerr,u):
end if:
#
# Fixed step RK Fehlberg (RKF45) integrator
if (ncase = 5) then
    read "c:\odelib\maple\pdenon\rkf45a.txt":
    rkf45a(neqn,t0,tf,u0,nsteps,u):
end if:
#
# Variable step RK Fehlberg (RKF45) integrator
if (ncase = 6) then
    read "c:\odelib\maple\pdenon\rkf45b.txt":
    rkf45b(neqn,t0,tf,u0,nsteps,abserr,relerr,u):
end if:
#
# Advance solution
    t0:=tf:
    tf:=tf+tp:
    for i from 1 to neqn do
        u0[i]:=u[i]:
    end do:
#
# Next output
end do:
#
# Next integrator
end do:
#
# End of pdenon
end:

Program 5.6.2
Maple main program pdenon.txt for the numerical integration of Equations 5.8 to 5.12
Note the reference to specific files by read statements, e.g.,

```
# # Initial condition vector
  read *c:\odelib\maple\pdenon\inital.txt*:
  inital(neqn,t0,u0):
```

**intpar, inital, derr, and fprint are listed below:**

```
intpar:=proc()
  # # Function intpar sets the parameters to control the 
  # integration of the nonlinear PDE problem 
  # # Type variables 
  global neqn, nout, nsteps, t0, tf, abserr, relerr:
  # # Number of first order ODEs 
  neqn:=22:
  # # Number of output points 
  nout:=6:
  # # Maximum number of steps in the interval t0 to tf 
  nsteps:=1000:
  # # Initial, final values of independent variable 
  t0:=0.0:
  tf:=0.2:
  # # Error tolerances 
  abserr:=1.0e-01:
  relerr:=1.0e-05:
  # # End of intpar
  end:
```

```
inital:=proc(neqn,t,u0)
  # # Function inital sets the initial condition vector 
  # of the nonlinear PDE problem 
  # # Type variables 
  local ui, i:
  # ```
# Problem parameters
ui:=298.0:
#
# Initial condition
for i from 1 to neqn do
  u0[i]:=ui:
end do:
#
# End of initial
end:

derv:=proc(neqn,t,u,ut)
#
# Function derv computes the derivative vector
# of the nonlinear PDE problem
#
# Type variables
local u0,
 dx, i,
 alpha, k, sigma,
 L, ui, ua,
 a, e, rhos,
 cps, ls, cs:

# Problem parameters
alpha:=1.0e-06: k:=1.0: sigma:=5.67e-08:
L:=0.1: ui:=298.0: ua:=2000.0:
a:=1.0: e:=1.0: rhos:=7800.0:
cps:=435.0: ls:=0.025: cs:=rhos*cps*ls:
#
# Spatial grid
dx:=1.0/(neqn-2):
dxs:=dx*dx:
#
# Insulation
for i from 1 to neqn-2 do
  if(i=1)then
    u0:=u[2]+2.0*dx*L*(sigma/k)*(a*ua^4-e*u[1]^4):
    ut[1]:=(u[2]-2.0*u[1]+u0)/dxs:
  else
    ut[i]:=(u[i+1]-2.0*u[i]+u[i-1])/dxs:
  end if:
end do:
# Steel
ut[neqn] := (k*L/alpha)*(1.0/cs)*(u[neqn-2]-u[neqn-1])/dx:
ut[neqn-1] := ut[neqn]:

# End of derv
end:

fprint:=proc(ncase,neqn,t,u)
# Function fprint displays the numerical solution
to the nonlinear PDE problem
#
# Declare global variables
global nsteps:
#
# Type variables
local im, tmin, dx, dxs, i,
alpha, k, sigma,
L, ui, ua,
a, e, rhos,
cps, ls, cs:
#
# Problem parameters
alpha:=1.0e-06: k:=1.0: sigma:=5.67e-08:
L:=0.1: ui:=298.0: ua:=2000.0:
a:=1.0: e:=1.0: rhos:=7800.0:
cps:=435.0: ls:=0.025: cs:=rhos*cps*ls:
#
# Print a heading for the solution at t = 0
if (t <= 0.0) then
    # Label for ODE integrator
    # Fixed step modified Euler
    if (ncase = 1) then
        printf(`\n\n euler2a integrator\n\n`);
    #
    # Variable step modified Euler
    elif (ncase = 2) then
        printf(`\n\n euler2b integrator\n\n`);
    #
Program 5.6.3

*intpar, initial, derv,* and *fprint* for the solution of Equations 5.8 to 5.12

Programs 5.6.1, 5.6.2, and 5.6.3 closely parallel Programs 4.6.1, 4.6.2, and 4.6.3 so they will not be discussed here.

The output from the preceding routines is listed below:


**euler2a integrator**

```plaintext
ncase = 1  neqn = 22  nsteps = 1000

<table>
<thead>
<tr>
<th>t</th>
<th>u(1)</th>
<th>u(im)</th>
<th>u(neqn)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>298.00</td>
<td>298.00</td>
<td>298.00</td>
</tr>
<tr>
<td>33.3</td>
<td>1987.96</td>
<td>1002.15</td>
<td>394.17</td>
</tr>
<tr>
<td>66.7</td>
<td>1991.05</td>
<td>1218.09</td>
<td>622.84</td>
</tr>
<tr>
<td>100.0</td>
<td>1992.55</td>
<td>1345.04</td>
<td>834.65</td>
</tr>
<tr>
<td>133.3</td>
<td>1993.72</td>
<td>1447.20</td>
<td>1015.31</td>
</tr>
<tr>
<td>166.7</td>
<td>1994.70</td>
<td>1533.03</td>
<td>1168.10</td>
</tr>
</tbody>
</table>
```

**euler2b integrator**

```plaintext
ncase = 2  neqn = 22  nsteps = 1000

<table>
<thead>
<tr>
<th>t</th>
<th>u(1)</th>
<th>u(im)</th>
<th>u(neqn)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>298.00</td>
<td>298.00</td>
<td>298.00</td>
</tr>
<tr>
<td>33.3</td>
<td>1987.93</td>
<td>1002.11</td>
<td>394.14</td>
</tr>
<tr>
<td>66.7</td>
<td>1991.00</td>
<td>1217.92</td>
<td>622.58</td>
</tr>
<tr>
<td>100.0</td>
<td>1992.54</td>
<td>1344.68</td>
<td>834.02</td>
</tr>
<tr>
<td>133.3</td>
<td>1993.68</td>
<td>1446.70</td>
<td>1014.43</td>
</tr>
<tr>
<td>166.7</td>
<td>1994.65</td>
<td>1532.29</td>
<td>1166.78</td>
</tr>
</tbody>
</table>
```

**rkc4a integrator**

```plaintext
ncase = 3  neqn = 22  nsteps = 1000

<table>
<thead>
<tr>
<th>t</th>
<th>u(1)</th>
<th>u(im)</th>
<th>u(neqn)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>298.00</td>
<td>298.00</td>
<td>298.00</td>
</tr>
<tr>
<td>33.3</td>
<td>1987.96</td>
<td>1002.16</td>
<td>394.17</td>
</tr>
<tr>
<td>66.7</td>
<td>1991.05</td>
<td>1218.10</td>
<td>622.85</td>
</tr>
<tr>
<td>100.0</td>
<td>1992.55</td>
<td>1345.05</td>
<td>834.66</td>
</tr>
<tr>
<td>133.3</td>
<td>1993.72</td>
<td>1447.20</td>
<td>1015.32</td>
</tr>
<tr>
<td>166.7</td>
<td>1994.70</td>
<td>1533.03</td>
<td>1168.10</td>
</tr>
</tbody>
</table>
```

**rkc4b integrator**

```plaintext
ncase = 4  neqn = 22  nsteps = 1000

<table>
<thead>
<tr>
<th>t</th>
<th>u(1)</th>
<th>u(im)</th>
<th>u(neqn)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>298.00</td>
<td>298.00</td>
<td>298.00</td>
</tr>
</tbody>
</table>
```
This output is equivalent to the output from the preceding MATLAB (Section 5.1), C (Section 5.2), C++ (Section 5.3), Fortran (Section 5.4), and Java (Section 5.5).

This completes the discussion of the nonlinear PDE problem programmed in the six languages. We now consider an interpretation of the application of the six programs and the outputs.

The collapse of the World Trade Center (WTC) towers on September 11, 2001 is generally attributed to the effect of the fires that resulted from burning jet fuel. Specifically, the structural steel was heated by the fires until it was weakened to the point that it could no longer support the towers; the four to six floors that suffered the most direct impact of the fires collapsed onto the
floors below, which in turn were overloaded and collapsed. This successive floor collapse propagated down the towers until they completely collapsed onto the ground.

The structural steel was insulated to protect it from fire (a standard practice in the construction of buildings with steel frames). The question of how well this insulation protected the columns from the fires is complicated by two additional factors:

- Some of the vertical columns in the outer walls of the towers were sheared off by the planes; thus, vertical support from these columns was lost.
- The insulation at the point of impact of the planes was damaged and, in the case of some columns, was dislodged so that the steel in those columns was unprotected from the fire.

Thus, the situation surrounding the impact of the planes and the resulting fires was complicated, and a detailed analysis is certainly beyond the scope of this discussion. However, a central question remains of how much the insulation that remained intact protected the steel columns from the burning jet fuel. The fact that the buildings did not collapse immediately suggests that the insulation was effective; the delayed collapse, summarized below, may have saved up to as many as 25,000 lives of the people who were able to escape the towers:

<table>
<thead>
<tr>
<th>Tower</th>
<th>Time to Collapse (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>North</td>
<td>143</td>
</tr>
<tr>
<td>South</td>
<td>56</td>
</tr>
</tbody>
</table>

The preceding model of steel covered with insulation and subject to a high temperature through thermal radiation gives some insight into how the insulation might have performed. The numerical parameters that were used in the calculations are representative of the values for steel, insulation, and thermal radiation, but they cannot be considered accurate in the sense that they apply with certainty to the situation in the towers immediately after the time of impact of the planes. This is particularly true because of the complexity of the devastation that occurred at the points of impact.

However, solutions to the model Equations 5.8 to 5.12 give an indication of how the structural steel temperatures may have increased with time. Specifically, if we consider the last output (from Programs 5.6.1, 5.6.2, and 5.6.3), we first observe that the temperature of the insulation directly facing the flame, \( u(1) \), increases rapidly:
This numerical output also gives some indication of what might have happened to the steel from which the insulation was dislodged. Clearly in a matter of a few minutes the temperatures reached values that would cause the steel to soften.

The interior insulation temperatures (at the midpoint of the insulation, \( u(\text{im}) = u(11) \), and at the insulation/steel interface, \( u(\text{neqn}) = u(22) \)) were substantially delayed in time:

<table>
<thead>
<tr>
<th>t</th>
<th>( u(1) )</th>
<th>( u(\text{im}) )</th>
<th>( u(\text{neqn}) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>298.00</td>
<td>298.00</td>
<td>298.00</td>
</tr>
<tr>
<td>33.3</td>
<td>1988.01</td>
<td>1002.16</td>
<td>394.17</td>
</tr>
<tr>
<td>66.7</td>
<td>1991.05</td>
<td>1217.81</td>
<td>622.40</td>
</tr>
<tr>
<td>100.0</td>
<td>1992.54</td>
<td>1344.61</td>
<td>833.89</td>
</tr>
<tr>
<td>133.3</td>
<td>1993.73</td>
<td>1446.47</td>
<td>1014.02</td>
</tr>
<tr>
<td>166.7</td>
<td>1994.70</td>
<td>1532.10</td>
<td>1166.45</td>
</tr>
</tbody>
</table>

At the insulation/steel interface (\( \text{neqn} = 22 \)), the steel temperature eventually enters a temperature range that would lead to substantial weakening of the steel,

<table>
<thead>
<tr>
<th>Steel Temperature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (min)</td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>66.7</td>
</tr>
<tr>
<td>100.0</td>
</tr>
<tr>
<td>133.3</td>
</tr>
</tbody>
</table>

The time delay of about 60 to 120 min is approximately in agreement with the time of collapse of the towers, which suggests that the insulation was effective in delaying the collapse.
However, there is substantial uncertainty in the model. Therefore, the reader may wish to study the effect of parameter changes on the model output. The parameter values are set in the following code taken from Program 5.6.3:

```
# Problem parameters
alpha:=1.0e-06: k:=1.0: sigma:=5.67e-08:
L:=0.1:      ui:=298.0:     ua:=2000.0:
a:=1.0:      e:=1.0:       rhos:=7800.0:
cps:=435.0:  ls:=0.025:     cs:=rhos*cps*ls:
```

Here are some suggested variations in the parameter values:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Current Value</th>
<th>Variation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thermal conductivity, $k$</td>
<td>1.0 J/s·m²·K</td>
<td>0.5–2.0</td>
</tr>
<tr>
<td>Thermal diffusivity, $a$</td>
<td>$10^{-6}$ m²/s</td>
<td>$0.8 \times 10^{-6}$–$1.2 \times 10^{-6}$</td>
</tr>
<tr>
<td>Insulation thickness, $L$</td>
<td>0.1 m</td>
<td>0.1–0.2</td>
</tr>
<tr>
<td>Steel thickness, $ls$</td>
<td>0.025 m</td>
<td>0.025–0.1</td>
</tr>
<tr>
<td>Absorptivity, $a$</td>
<td>1.0 (dimensionless)</td>
<td>0.25–1.0</td>
</tr>
<tr>
<td>Emissivity, $e$</td>
<td>1.0 (dimensionless)</td>
<td>0.25–1.0</td>
</tr>
<tr>
<td>Flame temperature, $ua$</td>
<td>2000 (K)</td>
<td>1800–2000</td>
</tr>
</tbody>
</table>

The reader will observe when making the suggested variations in the parameters that:

- Some parameters are more sensitive than others in affecting the steel temperature variations with time.
- The heating of the steel is substantially affected by the parameters. Thus, the conclusions drawn from the model (e.g., when the steel begins to weaken substantially) are determined significantly by the values of the model parameters.
- The preceding conclusion indicates the need for reliable parameter values, and the increased uncertainty of the conclusions drawn from the model with increased uncertainty in the model parameters. Thus, experimental measurements of the model parameters (in a laboratory) would be well worth the effort if models such as the one outlined in this chapter are to be used. Additionally, laboratory measurements of the strength of steel as a function of temperature would be essential for the interpretation of the solutions computed from the mathematical model of Equations 5.8 to 5.12.
- Although the uncertainty in the model output is substantial, we would look to a refinement of the model as the next logical step for better
understanding of the basic problem (the effect of fire on structural steel); that is, we would refine the equations and parameters. The alternative is to design without an analysis that serves as a quantitative guide; this would be unacceptable for the design of important physical and chemical systems.

Thus, we conclude with an indication of the following:

- How ODE/PDE models might be used in important applications
- Some numerical methods and associated software that can produce reliable (accurate) numerical solutions with reasonable effort

In general, computation has become an essential part of scientific and engineering analysis, particularly for situations that can only be studied mathematically, and perhaps augmented by laboratory data, such as the effect of fire on structural steel, and its implications for the safety of large structures. Another example is the growth of tumors modeled by the 1x1 ODE system. The insights gained by computational analysis are indispensable for the understanding and design of scientific and engineering systems. We hope the preceding material in Chapters 1 to 5, and the supplementary material in the following Appendices A to F, will be of assistance in this type of analysis.
The following 17 embedded RK pairs can be used for practice programming (Pairs 1, 9, and 11 are already programmed in Chapters 1 and 2 and the programs can therefore be used as templates). The performance of the error estimators can also be studied by applying the algorithms to problems with known analytical solutions as was done in Chapters 1 to 4. Of course, the 17 algorithms can also be applied to general \(nxn\) nonstiff ODE problems, with extensions to PDEs by the method of lines. Stiff ODEs can be studied by the methods discussed in Appendix C.

The equations of each algorithm are listed in the order in which they would be programmed; i.e., they are listed in an executable sequence. Three pairs appear in routines discussed in Chapter 2:

<table>
<thead>
<tr>
<th>Pair</th>
<th>Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>euler2a (fixed step)</td>
</tr>
<tr>
<td></td>
<td>euler2b (variable step)</td>
</tr>
<tr>
<td>9</td>
<td>rkf45a (fixed step)</td>
</tr>
<tr>
<td></td>
<td>rkf45b (variable step)</td>
</tr>
<tr>
<td>11</td>
<td>rk4a (fixed step)</td>
</tr>
<tr>
<td></td>
<td>rk4b (variable step)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pair 1</th>
<th>(1,2) Chap 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calculation</td>
<td>Equation</td>
</tr>
<tr>
<td>Stage 1</td>
<td>(k_1 = f(y_i, t_i)h)</td>
</tr>
<tr>
<td>Stage 2</td>
<td>(k_2 = f(y_i + k_1, t_i + h)h)</td>
</tr>
<tr>
<td>Step (O(h^1))</td>
<td>(y_{i+1}^{(1)} = y_i + k_1)</td>
</tr>
<tr>
<td>Est error</td>
<td>(e_{i+1}^{(1)} = \frac{1}{2}(k_2 - k_1))</td>
</tr>
<tr>
<td>Step (O(h^2))</td>
<td>(y_{i+1}^{(2)} = y_{i+1}^{(1)} + e_{i+1}^{(1)})</td>
</tr>
<tr>
<td>Step (t)</td>
<td>(t_{i+1} = t_i + h)</td>
</tr>
<tr>
<td>Pair 2</td>
<td>Calculation</td>
</tr>
<tr>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td>Stage 1</td>
<td>( k_1 = f(y_i, t_i)h )</td>
</tr>
<tr>
<td>Stage 2</td>
<td>( k_2 = f(y_i + \frac{2}{3} k_1, t_i + \frac{2}{3} h)h )</td>
</tr>
<tr>
<td>Stage 3</td>
<td>( k_3 = f(y_i + \frac{2}{3} k_2, t_i + \frac{2}{3} h)h )</td>
</tr>
<tr>
<td>Step ( O(h^2) )</td>
<td>( y_{i+1}^{(2)} = y_i + \frac{1}{2}k_1 + \frac{2}{3}k_2 )</td>
</tr>
<tr>
<td>Est error</td>
<td>( e_{i+1}^{(2)} = y_{i+1}^{(3)} - y_{i+1}^{(2)} )</td>
</tr>
<tr>
<td>Step ( t )</td>
<td>( t_{i+1} = t_i + h )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pair 3</th>
<th>Calculation</th>
<th>(2,3) Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stage 1</td>
<td>( k_1 = f(y_i, t_i)h )</td>
<td></td>
</tr>
<tr>
<td>Stage 2</td>
<td>( k_2 = f(y_i + \frac{2}{3} k_1, t_i + \frac{2}{3} h)h )</td>
<td></td>
</tr>
<tr>
<td>Stage 3</td>
<td>( k_3 = f(y_i - k_1 + 2k_2, t_i + h)h )</td>
<td></td>
</tr>
<tr>
<td>Step ( O(h^2) )</td>
<td>( y_{i+1}^{(2)} = y_i + \frac{1}{2}(k_1 + k_3) )</td>
<td></td>
</tr>
<tr>
<td>Est error</td>
<td>( e_{i+1}^{(2)} = \frac{1}{3}(-k_1 + 2k_2 - k_3) )</td>
<td></td>
</tr>
<tr>
<td>Step ( O(h^3) )</td>
<td>( y_{i+1}^{(3)} = y_{i+1}^{(2)} + e_{i+1}^{(2)} )</td>
<td></td>
</tr>
<tr>
<td>Step ( t )</td>
<td>( t_{i+1} = t_i + h )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pair 4</th>
<th>Calculation</th>
<th>(3,4) Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stage 1</td>
<td>( k_1 = f(y_i, t_i)h )</td>
<td></td>
</tr>
<tr>
<td>Stage 2</td>
<td>( k_2 = f(y_i + k_1, t_i + \frac{k}{3} h) )</td>
<td></td>
</tr>
<tr>
<td>Stage 3</td>
<td>( k_3 = f(y_i + \frac{1}{3} (k_1 + k_2), t_i + \frac{k}{3} h) )</td>
<td></td>
</tr>
<tr>
<td>Stage 4</td>
<td>( k_4 = f(y_i + \frac{2}{3} k_1 + \frac{1}{6} k_3, t_i + \frac{2}{3} h) )</td>
<td></td>
</tr>
<tr>
<td>Stage 5</td>
<td>( k_5 = f(y_i + \frac{1}{3} k_1 - \frac{1}{6} k_3 + 6k_4, t_i + h)h )</td>
<td></td>
</tr>
<tr>
<td>Step ( O(h^3) )</td>
<td>( y_{i+1}^{(3)} = y_i + \frac{1}{2}(k_1 + 4k_4 + k_3) )</td>
<td></td>
</tr>
<tr>
<td>Est error</td>
<td>( e_{i+1}^{(3)} = \frac{1}{3}(-k_1 + \frac{2}{3} k_3 - 4k_4 + \frac{1}{2} k_5) )</td>
<td></td>
</tr>
<tr>
<td>Step ( O(h^4) )</td>
<td>( y_{i+1}^{(4)} = y_{i+1}^{(3)} + e_{i+1}^{(3)} )</td>
<td></td>
</tr>
<tr>
<td>Step ( t )</td>
<td>( t_{i+1} = t_i + h )</td>
<td></td>
</tr>
<tr>
<td>Pair 5</td>
<td>Calculation</td>
<td>(3,4) Equation</td>
</tr>
<tr>
<td>-------</td>
<td>-------------</td>
<td>----------------</td>
</tr>
<tr>
<td>Stage 1</td>
<td>$k_1 = f(y_i, t_i)h$</td>
<td></td>
</tr>
<tr>
<td>Stage 2</td>
<td>$k_2 = f(y_i - 0.4k_1, t_i - 0.4h)h$</td>
<td></td>
</tr>
<tr>
<td>Stage 3</td>
<td>$k_3 = f(y_i + 0.6684895833k_1 - 0.2434895833k_2, ti + 0.425h)h$</td>
<td></td>
</tr>
<tr>
<td>Stage 4</td>
<td>$k_4 = f(y_i - 2.323685857k_1 + 1.125483359k_2 + 2.198202298k_3, t_i + h)h$</td>
<td></td>
</tr>
<tr>
<td>Step $O(h^3)$</td>
<td>$y_{i+1}^{(3)} = y_i + 0.0396825368k_2 + 0.7729468599k_3$</td>
<td>$+ 0.18737060041k_4$</td>
</tr>
<tr>
<td>Est error</td>
<td>$e_{i+1}^{(3)} = -y_{i+1}^{(3)} + y_i + 0.03431372549k_1 + 0.02705627706k_2$</td>
<td>$+ 0.7440130202k_3 + 0.1946169772k_4$</td>
</tr>
<tr>
<td>Step $O(h^4)$</td>
<td>$y_{i+1}^{(4)} = y_{i+1}^{(3)} + e_{i+1}^{(3)}$</td>
<td></td>
</tr>
<tr>
<td>Step $t$</td>
<td>$t_{i+1} = t_i + h$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pair 6</th>
<th>Calculation</th>
<th>(4,5) Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stage 1</td>
<td>$k_1 = f(y_i, t_i)h$</td>
<td></td>
</tr>
<tr>
<td>Stage 2</td>
<td>$k_2 = f(y_i + 0.0005k_1, t_i + 0.0005h)h$</td>
<td></td>
</tr>
<tr>
<td>Stage 3</td>
<td>$k_3 = f(y_i - 80.89939470k_1 + 81.18439470k_2, t_i + 0.285h)h$</td>
<td></td>
</tr>
<tr>
<td>Stage 4</td>
<td>$k_4 = f(y_i + 2113.327899k_1 - 2117.778035k_2 + 5.442136522k_3, t_i + 0.992h)h$</td>
<td></td>
</tr>
<tr>
<td>Stage 5</td>
<td>$k_5 = f(y_i + 2249.757677k_1 - 2254.489040k_2 + 5.739991965k_3 - 0.008629230728k_4, t_i + h)h$</td>
<td></td>
</tr>
<tr>
<td>Step $O(h^4)$</td>
<td>$y_{i+1}^{(4)} = y_i - 131.2823524k_1 + 131.4998223k_2 + 0.4837620276k_3 + 0.2987680554k_4$</td>
<td></td>
</tr>
<tr>
<td>Est error</td>
<td>$e_{i+1}^{(4)} = -y_{i+1}^{(4)} + y_i + 65.80784286k_1 - 65.94767173k_2 + 0.7959885276k_3 + 4.715404915k_4 - 4.371564570k_5$</td>
<td></td>
</tr>
<tr>
<td>Step $O(h^5)$</td>
<td>$y_{i+1}^{(5)} = y_{i+1}^{(4)} + e_{i+1}^{(4)}$</td>
<td></td>
</tr>
<tr>
<td>Step $t$</td>
<td>$t_{i+1} = t_i + h$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pair 7</th>
<th>Calculation</th>
<th>(3,4) Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stage 1</td>
<td>$k_1 = f(y_i, t_i)h$</td>
<td></td>
</tr>
<tr>
<td>Stage 2</td>
<td>$k_2 = f(y_i + \frac{1}{2}k_1, t_i + \frac{1}{2}h)h$</td>
<td></td>
</tr>
<tr>
<td>Stage 3</td>
<td>$k_3 = f(y_i + \frac{1}{2}(k_1 + k_2), t_i + \frac{1}{2}h)h$</td>
<td></td>
</tr>
<tr>
<td>Stage 4</td>
<td>$k_4 = f(y_i + \frac{1}{2}(k_1 + 3k_3), t_i + \frac{1}{2}h)h$</td>
<td></td>
</tr>
<tr>
<td>Stage 5</td>
<td>$k_5 = f(y_i + \frac{1}{2}k_1 - \frac{1}{2}k_3 + 2k_4, t_i + h)h$</td>
<td></td>
</tr>
<tr>
<td>Step $O(h^3)$</td>
<td>$y_{i+1}^{(3)} = y_i + \frac{1}{2}k_1 - \frac{1}{2}k_3 + 2k_4$</td>
<td></td>
</tr>
<tr>
<td>Est error</td>
<td>$e_{i+1}^{(3)} = -\frac{1}{2}k_1 + \frac{3}{2}k_3 - \frac{1}{2}k_4 + \frac{1}{2}k_5$</td>
<td></td>
</tr>
<tr>
<td>Step $O(h^4)$</td>
<td>$y_{i+1}^{(4)} = y_{i+1}^{(3)} + e_{i+1}^{(3)}$</td>
<td></td>
</tr>
<tr>
<td>Step $t$</td>
<td>$t_{i+1} = t_i + h$</td>
<td></td>
</tr>
</tbody>
</table>
### Pair 8 Calculation

<table>
<thead>
<tr>
<th>Stage 1</th>
<th>$k_1 = f(y_i, t_i)h$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stage 2</td>
<td>$k_2 = f(y_i + \frac{1}{2}k_1, t_i + \frac{1}{2}h)h$</td>
</tr>
<tr>
<td>Stage 3</td>
<td>$k_3 = f(y_i + \frac{1}{2}(k_1 + k_2), t_i + \frac{1}{2}h)h$</td>
</tr>
<tr>
<td>Stage 4</td>
<td>$k_4 = f(y_i - k_2 + 2k_3, t_i + h)h$</td>
</tr>
<tr>
<td>Stage 5</td>
<td>$k_5 = f(y_i + \frac{1}{3}(7k_1 + 10k_2 + k_4), t_i + \frac{2}{3}h)h$</td>
</tr>
<tr>
<td>Stage 6</td>
<td>$k_6 = f(y_i + \frac{1}{75}(28k_1 - 125k_2 + 546k_3 + 54k_4 - 378k_5), t_i + \frac{1}{2}h)h$</td>
</tr>
</tbody>
</table>

#### Calculation Equation

- **Step $O(h^4)$**
  
  $y_{i+1}^{(4)} = y_i + \frac{1}{5}(k_1 + 4k_3 + 4k_4)$

- **Est error**
  
  $e_{i+1}^{(4)} = \frac{3}{256}(-42k_1 - 224k_3 - 21k_4 + 162k_5 + 125k_6)$

- **Step $O(h^5)$**
  
  $y_{i+1}^{(5)} = y_{i+1}^{(4)} + e_{i+1}^{(4)}$

- **Step $t$**
  
  $t_{i+1} = t_i + h$

---

### Pair 9 Calculation

<table>
<thead>
<tr>
<th>Stage 1</th>
<th>$k_1 = f(y_i, t_i)h$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stage 2</td>
<td>$k_2 = f(y_i + \frac{1}{2}k_1, t_i + \frac{1}{2}h)h$</td>
</tr>
<tr>
<td>Stage 3</td>
<td>$k_3 = f(y_i + \frac{1}{2}(3k_1 + 9k_2), t_i + \frac{3}{2}h)h$</td>
</tr>
<tr>
<td>Stage 4</td>
<td>$k_4 = f(y_i + \frac{1}{27}(1932k_1 - 7200k_2 + 7296k_3), t_i + \frac{12}{13}h)h$</td>
</tr>
<tr>
<td>Stage 5</td>
<td>$k_5 = f(y_i + \frac{4288}{2025}k_1 - 8k_2 + \frac{3644}{2025}k_3 - \frac{844}{2025}k_4 - \frac{36}{2025}k_5, t_i + \frac{4}{3}h)h$</td>
</tr>
<tr>
<td>Stage 6</td>
<td>$k_6 = f(y_i + \frac{8}{49}k_1 + 2k_2 - \frac{3544}{3600}k_3 + \frac{1689}{3600}k_4 - \frac{121}{3600}k_5, t_i + \frac{1}{2}h)h$</td>
</tr>
</tbody>
</table>

#### Calculation Equation

- **Step $O(h^4)$**
  
  $y_{i+1}^{(4)} = y_i + \frac{25}{512}k_1 + \frac{140}{512}k_3 + \frac{25207}{3072}k_4 - \frac{3}{13}k_5$

- **Step $O(h^5)$**
  
  $y_{i+1}^{(5)} = y_i + \frac{16}{135}k_1 + \frac{5606}{5400}k_3 + \frac{2562}{5400}k_4 - \frac{9}{59}k_5 + \frac{3}{58}k_6$

- **Est error**
  
  $e_{i+1}^{(4)} = y_{i+1}^{(5)} - y_{i+1}^{(4)}$

- **Step $t$**
  
  $t_{i+1} = t_i + h$
### Pair 10 Calculation Equation

Stage 1

\[ k_1 = f(y_i, t_i)h \]

Stage 2

\[ k_2 = f(y_i + \frac{1}{2}k_1, t_i + \frac{1}{2}h)h \]

Stage 3

\[ k_3 = f(y_i + \frac{3}{4}k_1 + \frac{9}{40}k_2, t_i + \frac{1}{4}h)h \]

Stage 4

\[ k_4 = f(y_i + \frac{3}{4}k_1 - \frac{9}{40}k_2 + \frac{12}{5}k_3, t_i + \frac{1}{4}h)h \]

Stage 5

\[ k_5 = f(y_i - \frac{11}{12}k_1 + \frac{5}{6}k_2 - \frac{13}{12}k_3 + \frac{28}{15}k_4, t_i + h)h \]

Stage 6

\[ k_6 = f(y_i + \frac{163}{55296}k_1 + \frac{1828}{55296}k_2 + \frac{1275}{32768}k_3 + \frac{48275}{110592}k_4 + \frac{513}{1728}k_5, t_i + \frac{7}{8}h)h \]

Step \( O(h^4) \)

\[ y_{i+1}^{(4)} = y_i + \frac{2825}{720}k_1 + \frac{18575}{720}k_2 + \frac{1275}{32768}k_3 + \frac{48275}{110592}k_4 + \frac{513}{1728}k_5 + \frac{1}{4}k_6 \]

Step \( O(h^5) \)

\[ y_{i+1}^{(5)} = y_i + \frac{163}{55296}k_1 + \frac{1828}{55296}k_2 + \frac{1275}{32768}k_3 + \frac{48275}{110592}k_4 + \frac{513}{1728}k_5 + \frac{1}{4}k_6 \]

Est error

\[ e_{i+1}^{(4)} = y_{i+1}^{(5)} - y_{i+1}^{(4)} \]

Step \( t \)

\[ t_{i+1} = t_i + h \]

---

### Pair 11 Calculation Equation

Stage 1

\[ k_1 = f(y_i, t_i)h \]

Stage 2

\[ k_2 = f(y_i + \frac{1}{2}k_1, t_i + \frac{1}{2}h)h \]

Stage 3

\[ k_3 = f(y_i + \frac{1}{2}k_2, t_i + \frac{1}{2}h)h \]

Stage 4

\[ k_4 = f(y_i + k_3, t_i + h)h \]

Step \( O(h^2) \)

\[ y_{i+1}^{(2)} = y_i + k_2 \]

Est error

\[ e_{i+1}^{(2)} = \frac{1}{8}(k_1 - 4k_2 + 2k_3 + k_4) \]

Step \( O(h^4) \)

\[ y_{i+1}^{(4)} = y_{i+1}^{(2)} + e_{i+1}^{(2)} \]

Step \( t \)

\[ t_{i+1} = t_i + h \]
<table>
<thead>
<tr>
<th>Pair 12</th>
<th>Calculation</th>
<th>(2,4) Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stage 1</td>
<td>( k_1 = f(y_i, t_i)h )</td>
<td></td>
</tr>
<tr>
<td>Stage 2</td>
<td>( k_2 = f(y_i + \frac{1}{2}k_1, t_i + \frac{1}{2}h)h )</td>
<td></td>
</tr>
<tr>
<td>Stage 3</td>
<td>( k_3 = f(y_i - \frac{1}{2}k_1 + k_2, t_i + \frac{1}{2}h)h )</td>
<td></td>
</tr>
<tr>
<td>Stage 4</td>
<td>( k_4 = f(y_i + k_1 - k_2 + k_3, t_i + h)h )</td>
<td></td>
</tr>
<tr>
<td>Step ( O(h^2) )</td>
<td>( y_{i+1}^{(2)} = y_i + \frac{1}{2}k_1 + \frac{3}{2}k_2 )</td>
<td></td>
</tr>
<tr>
<td>Est error</td>
<td>( e_i^{(2)} = \frac{1}{6}(5k_1 - 9k_2 + 3k_3 + k_4) )</td>
<td></td>
</tr>
<tr>
<td>Step ( O(h^4) )</td>
<td>( y_{i+1}^{(4)} = y_{i+1}^{(2)} + e_i^{(2)} )</td>
<td></td>
</tr>
<tr>
<td>Step ( t )</td>
<td>( t_{i+1} = t_i + h )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pair 13</th>
<th>Calculation</th>
<th>(2,4) Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stage 1</td>
<td>( k_1 = f(y_i, t_i)h )</td>
<td></td>
</tr>
<tr>
<td>Stage 2</td>
<td>( k_2 = f(y_i + \frac{1}{2}k_1, t_i + \frac{1}{2}h)h )</td>
<td></td>
</tr>
<tr>
<td>Stage 3</td>
<td>( k_3 = f(y_i - \frac{1}{2}k_1 + \frac{1}{2}k_2, t_i + \frac{1}{2}h)h )</td>
<td></td>
</tr>
<tr>
<td>Stage 4</td>
<td>( k_4 = f(y_i - \frac{1}{2}k_1 + \frac{3}{2}k_2 + (1 + \frac{1}{\sqrt{2}})k_3, t_i + h)h )</td>
<td></td>
</tr>
<tr>
<td>Step ( O(h^2) )</td>
<td>( y_{i+1}^{(2)} = y_i + k_2 )</td>
<td></td>
</tr>
<tr>
<td>Est error</td>
<td>( e_i^{(2)} = \frac{1}{2}(2 + \frac{1}{\sqrt{2}})k_2 + 2(1 + \frac{1}{\sqrt{2}})k_3 + k_4) )</td>
<td></td>
</tr>
<tr>
<td>Step ( O(h^4) )</td>
<td>( y_{i+1}^{(4)} = y_{i+1}^{(2)} + e_i^{(2)} )</td>
<td></td>
</tr>
<tr>
<td>Step ( t )</td>
<td>( t_{i+1} = t_i + h )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pair 14</th>
<th>Calculation</th>
<th>(2,4) Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stage 1</td>
<td>( k_1 = f(y_i, t_i)h )</td>
<td></td>
</tr>
<tr>
<td>Stage 2</td>
<td>( k_2 = f(y_i + \frac{1}{2}k_1, t_i + \frac{1}{2}h)h )</td>
<td></td>
</tr>
<tr>
<td>Stage 3</td>
<td>( k_3 = f(y_i - \frac{1}{2}k_1 + k_2, t_i + \frac{1}{2}h)h )</td>
<td></td>
</tr>
<tr>
<td>Stage 4</td>
<td>( k_4 = f(y_i + \frac{1}{2}k_1 + \frac{3}{2}k_2 + k_3, t_i + h)h )</td>
<td></td>
</tr>
<tr>
<td>Step ( O(h^2) )</td>
<td>( y_{i+1}^{(2)} = y_i + k_2 )</td>
<td></td>
</tr>
<tr>
<td>Est error</td>
<td>( e_i^{(2)} = \frac{1}{6}(k_1 - 3k_2 + k_3 + k_4) )</td>
<td></td>
</tr>
<tr>
<td>Step ( O(h^4) )</td>
<td>( y_{i+1}^{(4)} = y_{i+1}^{(2)} + e_i^{(2)} )</td>
<td></td>
</tr>
<tr>
<td>Step ( t )</td>
<td>( t_{i+1} = t_i + h )</td>
<td></td>
</tr>
</tbody>
</table>
### Pair 15 (2,4)

<table>
<thead>
<tr>
<th>Calculation</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stage 1</td>
<td>$k_1 = f(y_i, t_i)h$</td>
</tr>
<tr>
<td>Stage 2</td>
<td>$k_2 = f(y_i + \frac{1}{2}k_1, t_i + \frac{1}{2}h)h$</td>
</tr>
<tr>
<td>Stage 3</td>
<td>$k_3 = f(y_i + 0.29697760924775360k_1 + 0.15875964497103583k_2, t_i + 0.45573725421878943h)h$</td>
</tr>
<tr>
<td>Stage 4</td>
<td>$k_4 = f(y_i + 0.21810038822592047k_1 - 3.0509651486929308k_2 + 3.8328647604670103k_3, t_i + h)h$</td>
</tr>
<tr>
<td>Step $O(h^2)$</td>
<td>$y_{i+1}^{(2)} = y_i - \frac{1}{4}k_1 + 1.25k_2$</td>
</tr>
<tr>
<td>Est error</td>
<td>$e_{i+1}^{(2)} = 0.42476028226269037k_1 - 1.8014806628787329k_2$ $+ 1.205535993965235k_3 + 0.17118478121951903k_4$</td>
</tr>
<tr>
<td>Step $O(h^4)$</td>
<td>$y_{i+1}^{(4)} = y_{i+1}^{(2)} + e_{i+1}^{(2)}$</td>
</tr>
<tr>
<td>Step $t$</td>
<td>$t_{i+1} = t_i + h$</td>
</tr>
</tbody>
</table>

### Pair 16 (2,5)

<table>
<thead>
<tr>
<th>Calculation</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stage 1</td>
<td>$k_1 = f(y_i, t_i)h$</td>
</tr>
<tr>
<td>Stage 2</td>
<td>$k_2 = f(y_i + \frac{1}{2}k_1, t_i + \frac{1}{2}h)h$</td>
</tr>
<tr>
<td>Stage 3</td>
<td>$k_3 = f(y_i + \frac{1}{4}(k_1 + k_2), t_i + \frac{1}{4}h)h$</td>
</tr>
<tr>
<td>Stage 4</td>
<td>$k_4 = f(y_i - k_2 + 2k_3, t_i + h)h$</td>
</tr>
<tr>
<td>Stage 5</td>
<td>$k_5 = f(y_i + \frac{1}{7}(7k_1 + 10k_2 + k_4), t_i + \frac{2}{3}h)h$</td>
</tr>
<tr>
<td>Stage 6</td>
<td>$k_6 = f(y_i + \frac{1}{126}(28k_1 - 125k_2 + 546k_3 + 54k_4 - 378k_5), t_i + \frac{1}{2}h)h$</td>
</tr>
<tr>
<td>Step $O(h^2)$</td>
<td>$y_{i+1}^{(2)} = y_i + \frac{1}{2}k_2$</td>
</tr>
<tr>
<td>Est error</td>
<td>$e_{i+1}^{(2)} = \frac{1}{306}(14k_1 - 336k_2 + 35k_4 + 162k_5 + 125k_6)$</td>
</tr>
<tr>
<td>Step $O(h^5)$</td>
<td>$y_{i+1}^{(5)} = y_{i+1}^{(2)} + e_{i+1}^{(2)}$</td>
</tr>
<tr>
<td>Step $t$</td>
<td>$t_{i+1} = t_i + h$</td>
</tr>
</tbody>
</table>
Pair 17 Calculation (2,5) Equation

Stage 1 \( k_1 = f(y_i, t_i)h \)

Stage 2 \( k_2 = f(y_i + \frac{1}{2}k_1, t_i + \frac{1}{2}h)h \)

Stage 3 \( k_3 = f(y_i + \frac{1}{32}(4k_1 + 6k_2), t_i + \frac{7}{3}h)h \)

Stage 4 \( k_4 = f(y_i + \frac{1}{2}(k_1 - 12k_2 + 3k_3), t_i + h)h \)

Stage 5 \( k_5 = f(y_i + \frac{1}{8}(6k_1 + 90k_2 - 50k_3 + 8k_4), t_i + \frac{5}{3}h)h \)

Stage 6 \( k_6 = f(y_i + \frac{1}{36}(6k_1 + 36k_2 + 10k_3 + 8k_4), t_i + \frac{1}{2}h)h \)

Step \( O(h^2) \) \( y_{i+1}^{(2)} = y_i - \frac{1}{2}k_1 + \frac{3}{2}k_2 \)

Est error \( e_{i+1}^{(2)} = \frac{1}{192}(119k_1 - 288k_2 + 125k_3 - 81k_5 + 125k_6) \)

Step \( O(h^3) \) \( y_{i+1}^{(3)} = y_{i+1}^{(2)} + e_{i+1}^{(2)} \)

Step \( t_{i+1} = t_i + h \)

Note that these 17 integrators have error estimates based on varying numbers of terms in the associated Taylor series. The number of terms in each error estimate, which just equals the difference in the orders of the pairs, is summarized below. For example, a \((2, 4)\) pair has \(4 - 2 = 2\) terms in the error estimate.

<table>
<thead>
<tr>
<th>Pairs</th>
<th>Terms in the Error Estimate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,2,3,4,5,6,7,8,9,10</td>
<td>1</td>
</tr>
<tr>
<td>11,12,13,14,15</td>
<td>2</td>
</tr>
<tr>
<td>16,17</td>
<td>3</td>
</tr>
</tbody>
</table>

The algorithms with 3 term error estimates (Pairs 16 and 17) are recommended for ODE problems with particularly stringent accuracy requirements.

In addition to the references for Pairs 1, 2, 9, and 11 given in Chapter 1, references for the other pairs (except 10), are given in Reference 1. This reference also provides Fortran routines for the ODE integrators summarized above (except 2 and 10). Pair 10 is taken from Reference 2.

References

Appendix B

Integrals from ODEs

The ODE algorithms discussed in Chapter 1 can be used to compute one-dimensional integrals. Specifically, the special case ODE (with the derivative function \( f(t) \) a function of the independent variable \( t \) and not the dependent variable \( y \))

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

(B.1)

has the solution

\[
y(t) = y_0 + \int_{t_0}^{t} f(\tau) d\tau
\]

(B.2)

Thus, we can integrate Equation B.1 to produce a numerical value of the integral of Equation B.2. To illustrate this procedure, consider the ODE

\[
d\frac{y}{dt} = \frac{2}{\sqrt{\pi}} e^{-t^2}, \quad y(0) = 0
\]

(B.3)

which has the solution

\[
y(t) = \frac{2}{\sqrt{\pi}} \int_{0}^{t} e^{-\tau^2} d\tau
\]

(B.4)

\( y(t) \) of Equation B.4 is the well-known error function, \( erf(t) \), which has broad application in science and engineering, and is tabulated extensively.

Functions \textit{intpar, initial, derv}, and \textit{fprint}, which follow directly from the preceding discussion (for the solution of Equation B.3), are listed below:

function [neqn,nout,nsteps,t0,tf,abserr,relerr]=intpar

% Function intpar sets the parameters to control the
% integration of the ODE with erf as a solution
%
% Number of first order ODEs
neqn=1;
% Number of output points
nout=6;
%
% Maximum number of steps in the interval t0 to tf
nsteps=100;
%
% Initial, final values of independent variable
t0=0.0;
tf=0.1;
%
% Error tolerances
abserr=1.0e-05;
relerr=1.0e-05;

function [y]=inital(neqn,t)
% Function inital sets the initial condition vector
% for the ODE with erf as the solution
% Initial condition
y(1)=0.0;

function [yt]=derv(neqn,t,y)
% Function derv computes the derivative vector
% of the ODE with erf as the solution
% Declare global variables
global nsteps;
% ODE
yt(1)=exp(-t^2);

function [out]=fprint(ncase,neqn,t,y)
% Function fprint displays the numerical and
% tabulated solutions to the ODE with erf as
% a solution
% Define global variables
global nsteps;
% Return current value of independent variable
% (MATLAB requires at least one return argument)
out=t;
%
% Print a heading for the solution at t = 0
if(t<=0.0)
%
% Label for ODE integrator
%
% Fixed step modified Euler
if(ncase==1)
  fprintf('
\n euler2a integrator
\n');
%
% Variable step modified Euler
elseif(ncase==2)
  fprintf('
\n euler2b integrator
\n');
%
% Fixed step classical fourth order RK
elseif(ncase==3)
  fprintf('
\n rkc4a integrator
\n');
%
% Variable step classical fourth order RK
elseif(ncase==4)
  fprintf('
\n rkc4b integrator
\n');
%
% Fixed step RK Fehlberg 45
elseif(ncase==5)
  fprintf('
\n rkf45a integrator
\n');
%
% Variable step RK Fehlberg 45
elseif(ncase==6)
  fprintf('
\n rkf45b integrator
\n');
end
%
% Heading
fprintf(' ncase = %2d nsteps = %3d
',ncase,nsteps);
fprintf(' t y(num) y(tab) diff\n');
%
% End of t = 0 heading
end
%
% Tabulated solution
if(t<0.001) ytab=0.0; end
if((t>0.0999)&(t<0.1001)) ytab=0.112463; end
if((t>0.1999)&(t<0.2001)) ytab=0.222703; end
if((t>0.2999)&(t<0.3001)) ytab=0.328627; end
if((t>0.3999)&(t<0.4001)) ytab=0.428392; end
if((t>0.4999)&(t<0.5001)) ytab=0.520500; end

% Numerical and tabulated solutions and their difference
% as output
    y1=(2.0/sqrt(pi))*y(1);
    diff=y1-ytab;
%
% Display the numerical and exact solutions, and their
% difference
    fprintf('%5.2f %11.6f %11.6f %11.6f
',t,y1,ytab,diff);

Program B.1
intpar, initial, derv, and fprintf for the solution of Equation B.3

The main program is again Program 3.1.1. The output from the preceding
program is listed below

euler2a integrator

ncase = 1 nsteps = 100

<table>
<thead>
<tr>
<th>t</th>
<th>y(num)</th>
<th>y(tab)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
</tr>
<tr>
<td>0.10</td>
<td>0.112463</td>
<td>0.112463</td>
<td>0.000000</td>
</tr>
<tr>
<td>0.20</td>
<td>0.222700</td>
<td>0.222700</td>
<td>0.000000</td>
</tr>
<tr>
<td>0.30</td>
<td>0.328627</td>
<td>0.328627</td>
<td>0.000000</td>
</tr>
<tr>
<td>0.40</td>
<td>0.428392</td>
<td>0.428392</td>
<td>0.000000</td>
</tr>
<tr>
<td>0.50</td>
<td>0.520500</td>
<td>0.520500</td>
<td>0.000000</td>
</tr>
</tbody>
</table>

euler2b integrator

ncase = 2 nsteps = 100

<table>
<thead>
<tr>
<th>t</th>
<th>y(num)</th>
<th>y(tab)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
</tr>
<tr>
<td>0.10</td>
<td>0.112461</td>
<td>0.112463</td>
<td>-0.000002</td>
</tr>
<tr>
<td>0.20</td>
<td>0.222700</td>
<td>0.222703</td>
<td>-0.000003</td>
</tr>
<tr>
<td>0.30</td>
<td>0.328623</td>
<td>0.328627</td>
<td>-0.000004</td>
</tr>
<tr>
<td>0.40</td>
<td>0.428388</td>
<td>0.428392</td>
<td>-0.000004</td>
</tr>
<tr>
<td>0.50</td>
<td>0.520496</td>
<td>0.520500</td>
<td>-0.000004</td>
</tr>
</tbody>
</table>
### rkc4a integrator

\( ncase = 3 \quad nsteps = 100 \)

<table>
<thead>
<tr>
<th>t</th>
<th>y(num)</th>
<th>y(tab)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
</tr>
<tr>
<td>0.10</td>
<td>0.112463</td>
<td>0.112463</td>
<td>0.000000</td>
</tr>
<tr>
<td>0.20</td>
<td>0.222703</td>
<td>0.222703</td>
<td>0.000000</td>
</tr>
<tr>
<td>0.30</td>
<td>0.328627</td>
<td>0.328627</td>
<td>0.000000</td>
</tr>
<tr>
<td>0.40</td>
<td>0.428392</td>
<td>0.428392</td>
<td>0.000000</td>
</tr>
<tr>
<td>0.50</td>
<td>0.520500</td>
<td>0.520500</td>
<td>0.000000</td>
</tr>
</tbody>
</table>

### rkc4b integrator

\( ncase = 4 \quad nsteps = 100 \)

<table>
<thead>
<tr>
<th>t</th>
<th>y(num)</th>
<th>y(tab)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
</tr>
<tr>
<td>0.10</td>
<td>0.112463</td>
<td>0.112463</td>
<td>0.000000</td>
</tr>
<tr>
<td>0.20</td>
<td>0.222703</td>
<td>0.222703</td>
<td>0.000000</td>
</tr>
<tr>
<td>0.30</td>
<td>0.328627</td>
<td>0.328627</td>
<td>0.000000</td>
</tr>
<tr>
<td>0.40</td>
<td>0.428392</td>
<td>0.428392</td>
<td>0.000000</td>
</tr>
<tr>
<td>0.50</td>
<td>0.520500</td>
<td>0.520500</td>
<td>0.000000</td>
</tr>
</tbody>
</table>

### rkf45a integrator

\( ncase = 5 \quad nsteps = 100 \)

<table>
<thead>
<tr>
<th>t</th>
<th>y(num)</th>
<th>y(tab)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
</tr>
<tr>
<td>0.10</td>
<td>0.112463</td>
<td>0.112463</td>
<td>0.000000</td>
</tr>
<tr>
<td>0.20</td>
<td>0.222703</td>
<td>0.222703</td>
<td>0.000000</td>
</tr>
<tr>
<td>0.30</td>
<td>0.328627</td>
<td>0.328627</td>
<td>0.000000</td>
</tr>
<tr>
<td>0.40</td>
<td>0.428392</td>
<td>0.428392</td>
<td>0.000000</td>
</tr>
<tr>
<td>0.50</td>
<td>0.520500</td>
<td>0.520500</td>
<td>0.000000</td>
</tr>
</tbody>
</table>

### rkf45b integrator

\( ncase = 6 \quad nsteps = 100 \)

<table>
<thead>
<tr>
<th>t</th>
<th>y(num)</th>
<th>y(tab)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
</tr>
<tr>
<td>0.10</td>
<td>0.112463</td>
<td>0.112463</td>
<td>0.000000</td>
</tr>
</tbody>
</table>
The error tolerances set in intpar are satisfied by all six integrators. Of course, this problem tests only the stepping in $t$, and not $y$, because of the special form of Equation B.1.
Appendix C

Stiff ODE Integration

The explicit integrators discussed in Chapters 1 and 2, and applied to ODE and PDE systems in Chapters 2 to 5, can be used for a broad spectrum of applications. However, as discussed in Section 1.7, these explicit (nonstiff) integrators will require lengthy calculations for stiff systems. Again, this difficulty with stiff systems results from having to take small integration steps to maintain stability of the numerical integration (because of the stability constraints of explicit algorithms) while having to cover a large interval in the independent variable to compute a complete solution. An example is the 2x2 system of Chapter 3 for which the maximum integration step to maintain stability is set by the largest eigenvalue, while the timescale of the solution is set by the smallest eigenvalue. If the eigenvalues are widely separated, the problem is stiff, and an implicit integrator is required to compute a solution efficiently, as we will demonstrate.

We now illustrate some of the properties of stiff systems using the 2x2 system. Specifically, we will consider:

- The BDF formulas of Section 1.7 applied to the 2x2 ODE system
- A MATLAB program for the first-order BDF method (implicit Euler method)
- The 2x2 ODE system integrated by the MATLAB stiff integrators `ode23s` and `ode15s`

C.1 The BDF Formulas Applied to the 2x2 ODE System

The BDF method is first applied to the general (possibly nonlinear) 2x2 ODE system

\[
\begin{align*}
\frac{dy_1}{dt} &= f_1(y_1, y_2, t) \\
\frac{dy_2}{dt} &= f_2(y_1, y_2, t)
\end{align*}
\]

(C.1)
The BDF equations are

\[ \alpha_0 y_{i+1,1} + \alpha_1 y_{i,1} + \cdots + \alpha_v y_{i-v+1,1} = h f_1(y_{i+1,1}, y_{i+1,2}, t_{i+1}) \]
\[ \alpha_0 y_{i+1,2} + \alpha_1 y_{i,2} + \cdots + \alpha_v y_{i-v+1,2} = h f_2(y_{i+1,1}, y_{i+1,2}, t_{i+1}) \]

or

\[ h f_1(y_{i+1,1}, y_{i+1,2}, t_{i+1}) - [\alpha_0 y_{i+1,1} + \alpha_1 y_{i,1} + \cdots + \alpha_v y_{i-v+1,1}] = g_1(y_{i+1,1}, y_{i+1,2} t_{i+1}) = 0 \]
\[ h f_2(y_{i+1,1}, y_{i+1,2}, t_{i+1}) - [\alpha_0 y_{i+1,2} + \alpha_1 y_{i,2} + \cdots + \alpha_v y_{i-v+1,2}] = g_2(y_{i+1,1}, y_{i+1,2}, t_{i+1}) = 0 \]  
(C.2)

Equations C.2 are a 2x2 nonlinear system (nonlinear if the original ODE system, Equations C.1, is nonlinear), for the two unknowns \( y_{i+1,1}, y_{i+1,2} \) (the ODE solution at the advanced point \( i+1 \)). We need to apply a nonlinear solver to Equations C.2 such as Newton’s method to compute the two unknowns, \( y_{i+1,1}, y_{i+1,2} \).

\[ J \delta y = -g(y) \]  
(C.3)

where for the nxn problem

\[ y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \quad \delta y = \begin{bmatrix} \delta y_1 \\ \delta y_2 \\ \vdots \\ \delta y_n \end{bmatrix} \]  
(C.4)(C.5)

\[ g = \begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ g_n \end{bmatrix} = \begin{bmatrix} g_1(y_1, y_2, \cdots y_n, t) \\ g_2(y_1, y_2, \cdots y_n, t) \\ \vdots \\ g_n(y_1, y_2, \cdots y_n, t) \end{bmatrix} \]  
(C.6)

\[ J = \begin{bmatrix} \frac{\partial g_1}{\partial y_1} & \frac{\partial g_1}{\partial y_2} & \cdots & \frac{\partial g_1}{\partial y_n} \\ \frac{\partial g_2}{\partial y_1} & \frac{\partial g_2}{\partial y_2} & \cdots & \frac{\partial g_2}{\partial y_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial g_n}{\partial y_1} & \frac{\partial g_n}{\partial y_2} & \cdots & \frac{\partial g_n}{\partial y_n} \end{bmatrix} = \begin{bmatrix} f_{11} & f_{12} & \cdots & f_{1n} \\ f_{21} & f_{22} & \cdots & f_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ f_{n1} & f_{n2} & \cdots & f_{nn} \end{bmatrix} \]  
(C.7)
**J** is the nxn Jacobian matrix, consisting of all first-order partial derivatives of the functions, \([g_1 \ g_2 \ldots g_i \ldots g_n]^T\) with respect to the dependent variables \([y_1 \ y_2 \ldots y_j \ldots y_n]^T\), i.e.,

\[
J_{ij} = \frac{\partial g_i}{\partial y_j}
\]  
(C.8)

\(\delta y\) is the vector of Newton corrections, which should decrease below a specified tolerance or threshold as the Newton iteration proceeds.

Application of the preceding equations to the 2x2 ODE system of Equations 1.6 of Chapter 1:

\[
\begin{align*}
\frac{dy_1}{dt} &= -ay_1 + by_2 \\
\frac{dy_2}{dt} &= by_1 - ay_2
\end{align*}
\]  
(C.9)

gives

\[
\begin{bmatrix} y_1 \\ y_2 \end{bmatrix}, \quad \delta y = \begin{bmatrix} \delta y_1 \\ \delta y_2 \end{bmatrix}
\]  
(C.10)(C.11)

\[
g = \begin{bmatrix} g_1(y_1, y_2, t) \\ g_2(y_1, y_2, t) \end{bmatrix}
\]

\[
= \begin{bmatrix} h[-ay_{i+1,1} + by_{i+1,2}] - [\alpha_0 y_{i+1,1} + \alpha_1 y_{i,1} + \ldots + \alpha_v y_{i-v+1,1}] \\ h[+by_{i+1,1} - ay_{i+1,2}] - [\alpha_0 y_{i+1,2} + \alpha_1 y_{i,2} + \ldots + \alpha_v y_{i-v+1,2}] \end{bmatrix}
\]  
(C.12)

\[
J = \begin{bmatrix} \frac{\partial g_1}{\partial y_1} & \frac{\partial g_1}{\partial y_2} \\ \frac{\partial g_2}{\partial y_1} & \frac{\partial g_2}{\partial y_2} \end{bmatrix} = \begin{bmatrix} J_{11} & J_{12} \\ J_{21} & J_{22} \end{bmatrix} = \begin{bmatrix} -ah - \alpha_0 & bh \\ bh & -ah - \alpha_0 \end{bmatrix}
\]  
(C.13)

Thus, Equation C.3 becomes

\[
\begin{bmatrix} -ah - \alpha_0 & bh \\ bh & -ah - \alpha_0 \end{bmatrix} \begin{bmatrix} \delta y_1 \\ \delta y_2 \end{bmatrix}
\]

\[
= \begin{bmatrix} h[-ay_{i+1,1} + by_{i+1,2}] - [\alpha_0 y_{i+1,1} + \alpha_1 y_{i,1} + \ldots + \alpha_v y_{i-v+1,1}] \\ h[+by_{i+1,1} - ay_{i+1,2}] - [\alpha_0 y_{i+1,2} + \alpha_1 y_{i,2} + \ldots + \alpha_v y_{i-v+1,2}] \end{bmatrix}
\]  
(C.14)

For the first-order BDF (first-order implicit RK or implicit Euler method), \(v = 1, \alpha_0 = 1, \alpha_1 = -1, \alpha_2, \ldots, \alpha_v = 0\). Thus, Equations C.14 require only the preceding values \(y_{i,1}, y_{i,2}\) (generally RK methods are self-starting while higher-order BDF methods require a history of the solution at the points \(i-1, i-2, \ldots i-v+1\)
to take the step from $i$ to $i + 1$; but, of course, the higher-order BDF methods have better accuracy than the first-order BDF.

Equations C.14 are programmed in the following MATLAB program. The numerical integration continues until the condition

\[
|\delta y_1| < \text{eps} \\
|\delta y_2| < \text{eps}
\]

is satisfied where $\text{eps}$ is a tolerance set in the program.

C.2 MATLAB Program for the Solution of the 2x2 ODE System

Equations C.14 with $\nu = 1$ (first-order BDF or the implicit Euler method) are solved by Newton’s method in the following program:

```matlab
% 2 x 2 ODE system by first order BDF

% Open a file for output
fid=fopen('appc1.out','w');

% Number of ODEs
neqn=2;

% Convergence tolerance, maximum number of iterations to compute Newton corrections
eps=0.00001;
maxiter=20;

% Initial condition
y10=0.0;
y20=2.0;
y(1)=y10;
y(2)=y20;
t=0.0;

% Values just for initial output
dy(1)=0.0;
dy(2)=0.0;
niter=0;

% Integration step, steps/output, number of outputs
h=0.01;
```
nsteps=100;
nout=11;
%
% Problem parameters
a=500000.5;
b=499999.5;
%
% Jacobian matrix
J(1,1)=-a*h-1.0;
J(1,2)=b*h;
J(2,1)=b*h;
J(2,2)=-a*h-1.0;
%
% Print a heading
fprintf( ' t dy(1) dy(2)
y(1) y(2) erry(1) erry(2) iter\n');
%
% nout outputs
for i=1:nout
  % Initial output or after nstep integration steps
  % completed; display Newton corrections, numerical and
  % exact solutions for output
  lambda1=-(a-b);
  lambda2=-(a+b);
  exp1=exp(lambda1*t);
  exp2=exp(lambda2*t);
  y1e=exp1-exp2;
  y2e=exp1+exp2;
  erry1=y1e-y(1);
  erry2=y2e-y(2);
  fprintf(fid,'%10.5f%10.5f%10.5f%10.5f%10.5f%10.5f
  %10.5f%5d\n',...t,dy(1),dy(2),y(1),y(2),erry1,erry2,niter);
  fprintf(fid,'%10.5f%10.5f%10.5f%10.5f%10.5f%10.5f
  %10.5f%5d\n',...t,dy(1),dy(2),y(1),y(2),erry1,erry2,niter);
%
% nsteps steps/output
for is=1:nsteps
  % Initialize iteration counter
  niter=1;
  stop=0;
  %
% Test for the end of the current step
while stop==0

% Functions g1, g2
% g = [h*(-a*y(1)+b*y(2))-y(1)+y10
%     h*( b*y(1)-a*y(2))-y(2)+y20];

% Solve for Newton corrections
% Gaussian elimination
dy=-J\g;

% Jacobian inverse
dy=-J^(-1)*g;

% Update solution
for ic=1:neqn
    y(ic)=y(ic)+dy(ic);
end
stop=1;

% Check if the corrections are within the tolerance eps
for ic=1:neqn
    if abs(dy(ic))>eps
        % Convergence not achieved; continue calculation
        niter=niter+1;
        stop=0;
        break;
    end
end

% If maximum iterations reached, accept current step
if (niter==maxiter) stop=1; end

% Continue integration step
end

% Integration step completed
y10=y(1);
y20=y(2);
t=t+h;

% Continue nstep integration steps
end
Program C.1
Solution of Equations C.14 by Newton’s method

We can note the following points about Program C.1:

• An output file is defined, then the number of nonlinear equations to be solved, in this case 2 for Equations C.14:

```matlab
% 2 x 2 ODE system by first order BDF
% Open a file for output
fid=fopen('appc1.out','w');
% Number of ODEs
neqn=2;
```

• The tolerance for the Newton corrections and the maximum number of Newton iterations are then defined numerically:

```matlab
% Convergence tolerance, maximum number of
% iterations to compute Newton corrections
eps=0.00001;
maxiter=20;
```

• An initial estimate of the solution, required by Newton’s method is defined, which is taken as the initial condition for Equations C.9 \( y_1(0) = 0, \ y_2(0) = 2 \). Also, the independent variable, \( t \), in Equations C.9 is initialized:

```matlab
% Initial condition
y10=0.0;
y20=2.0;
y(1)=y10;
y(2)=y20;
t=0.0;
```

• The Newton corrections are zeroed and the counter for the Newton iterations is initialized:
% Values just for initial output
dy(1)=0.0;
dy(2)=0.0;
niter=0;

• The variables that control the BDF integration are set, specifically, the integration step in Equations C.14 (this will be for a fixed step BDF integration), the maximum number of integration steps for each output interval, and the number of outputs:

% Integration step, steps/output, number of outputs
h=0.01;
nsteps=100;
nout=11;

• The parameters in Equations C.9 are set:

% Problem parameters
a=500000.5;
b=499999.5;

These values correspond to the stiff case discussed in Section 1.7 for which the eigenvalues of Equations 1.6 (or C.9) are $\lambda_1 = -(a - b) = -1$, $\lambda_2 = -(a + b) = -1, 000, 000$. Thus, there will be a total of $100 \times (11 - 1) = 1000$ Newton steps, each of length 0.01 so that the final value of $t$ is $0.01 \times 1000 = 10$.

• The elements of the $n\times n = 2\times2$ Jacobian matrix of Equation C.13 are programmed:

% Jacobian matrix
J(1,1)=-a*h-1.0;
J(1,2)=b*h;
J(2,1)=b*h;
J(2,2)=-a*h-1.0;

Note that since Equations C.9 are linear (constant coefficient ODEs), the Jacobian matrix is constant and therefore has to be evaluated only once. More generally, if the ODEs are nonlinear, the Jacobian matrix would have to be updated at each Newton iteration (which is a major portion of the calculation in using Newton’s method).
A heading for the numerical solution is then printed:

```matlab
% Print a heading
fprintf( ' t dy(1) dy(2) y(1) y(2) erry(1) erry(2) iter\n');
```

(this output statement has been put in two lines to fit within the available printed space; they should be returned to single lines if Equation C.1 is executed).

An outer loop is used to output the solution at the `nout` output points:

```matlab
% nout outputs
for i=1:nout
  % Initial output or after nstep integration steps
  % and exact solutions for output
  lambda1=-(a-b);
  lambda2=-(a+b);
  exp1=exp(lambda1*t);
  exp2=exp(lambda2*t);
  y1e=exp1-exp2;
  y2e=exp1+exp2;
  erry1=y1e-y(1);
  erry2=y2e-y(2);
  fprintf(fid,'%10.5f%10.5f%10.5f%10.5f%10.5f%10.5f
%10.5f%5d\n',...t,dy(1),dy(2),y(1),y(2),erry1,erry2,niter);
  fprintf( '%10.5f%10.5f%10.5f%10.5f%10.5f%10.5f
%10.5f%5d\n',...t,dy(1),dy(2),y(1),y(2),erry1,erry2,niter);
```

The first task in this loop is to compute the exact solution to Equations 1.6 (or Equations C.9), i.e., Equations 1.17. The difference between the numerical and exact solution is then computed, followed by printing of the independent variable, `t`, the two Newton corrections, the numerical solution, the error in the numerical solution, and the number of iterations required to produce the numerical solution.

An intermediate loop performs the calculations through `nsteps` integration steps of length `h`; the iteration counter is initialized for each integration step and a variable is initialized, which will indicate when the Newton iterations are stopped:
for is=1:nsteps
% Initialize iteration counter
niter=1;
stop=0;

• An inner loop then performs the Newton iterations while nstop= 0; first the vector of functions to be zeroed is computed according to Equation C.12 (with ν = 1):

% Test for the end of the current step
while stop==0
% Functions g1, g2
g=[h*(-a*y(1)+b*y(2))-y(1)+y10
    h*( b*y(1)-a*y(2))-y(2)+y20];

• The linear Newton Equations C.3 are then solved for the Newton corrections:

% Solve for Newton corrections
% Gaussian elimination
dy=-J\g;
% Jacobian inverse
dy=-J^(-1)*g;

Note that two methods of solution are programmed:
— Gaussian elimination, which is the preferred method of solution
— Inverse Jacobian matrix, i.e.,

\[ \delta y = -J^{-1}g(y) \] (C.15)

Although Equation C.15 is formally (mathematically) correct, it is not used in practice because computing the inverse Jacobian matrix is inefficient; rather, some form of Gaussian elimination is generally used. Note, however, how easily either method is programmed in MATLAB; this is due to the facility of MATLAB to handle matrices (arrays) without subscripting, plus the definition of basic matrix operations, e.g., multiplication, inverse, Gaussian reduction. This step (for computing the Newton corrections) will fail if the Jacobian matrix is singular or near singular (i.e., ill-conditioned). Thus, the calculation of the condition of J at
this point would be a good idea. MATLAB has a utility for calculating the condition of a matrix which can then be compared with the machine epsilon discussed in Section 1.7; specifically, if the condition number exceeds the reciprocal of the machine epsilon, the linear algebraic system (in this case, Equation C.3) is numerically singular.

• The Newton corrections are then applied to the current solution vector to produce (one hopes) an improved solution:

```matlab
% Update solution
for ic=1:neqn
    y(ic)=y(ic)+dy(ic);
end
stop=1;
```

If the Newton corrections are small enough (to be tested next), the iterations are terminated by setting \( stop = 1 \).

• Each Newton correction is tested against the convergence tolerance:

```matlab
% Check if the corrections are within the tolerance eps
for ic=1:neqn
    if abs(dy(ic))>eps
        % Convergence not achieved; continue calculation
        niter=niter+1;
        stop=0;
        break;
    end
end
```

If any Newton correction exceeds the tolerance, the iteration counter is incremented, the iterations are continued (\( stop = 0 \)) and the testing is ended (\( break \) from the \( for \) loop).

• If the maximum number of iterations is reached, the iterations are stopped. Otherwise, if the iterations are to be continued (\( stop = 0 \)), the next pass through the \( while \) loop (based on \( stop == 0 \)) is initiated.

```matlab
% If maximum iterations reached, accept current step
if (niter==maxiter) stop=1; end
%
% Continue integration step
end
```
Convergence has been achieved ($stop = 1$) so the new solution is now used as the old (base) solution in the next step of the ODE integration; the independent variable is incremented for the next step along the solution:

\[
\%
\text{Integration step completed}
\]
\[
y10 = y(1);
\]
\[
y20 = y(2);
\]
\[
t = t + h;
\]

- Finally, the *nstep* integration steps are completed, and the next output interval is covered until all *nout* output intervals are completed.

\[
\%
\text{Continue nstep integration steps}
\]
\[
end
\]
\[
\%
\text{Continue integration for next output interval}
\]
\[
end
\]

Note that Program C.1 contains a general Newton solver that can be applied to an $nxn$ system of nonlinear equations; all that really is required is to reprogram: (1) the Jacobian matrix (which generally will be inside the loop for the Newton iterations rather than outside) and (2) the vector of functions to be zeroed. Also, some tuning of the parameters will generally be required (e.g., the tolerance $\epsilon$ and the maximum number of iterations $\text{maxiter}$).

The output from the preceding program is as follows:

<table>
<thead>
<tr>
<th>$t$</th>
<th>$dy(1)$</th>
<th>$dy(2)$</th>
<th>$y(1)$</th>
<th>$y(2)$</th>
<th>$erry(1)$</th>
<th>$erry(2)$</th>
<th>iter</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00000</td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.00000</td>
<td>2.00000</td>
<td>0.00000</td>
<td>0.00000</td>
<td>0</td>
</tr>
<tr>
<td>1.00000</td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.36971</td>
<td>0.36971</td>
<td>-0.00183</td>
<td>-0.00183</td>
<td>2</td>
</tr>
<tr>
<td>2.00000</td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.13669</td>
<td>0.13669</td>
<td>-0.00135</td>
<td>-0.00135</td>
<td>2</td>
</tr>
<tr>
<td>3.00000</td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.05053</td>
<td>0.05053</td>
<td>-0.00075</td>
<td>-0.00075</td>
<td>2</td>
</tr>
<tr>
<td>4.00000</td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.01868</td>
<td>0.01868</td>
<td>-0.00037</td>
<td>-0.00037</td>
<td>2</td>
</tr>
<tr>
<td>5.00000</td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.00691</td>
<td>0.00691</td>
<td>-0.00017</td>
<td>-0.00017</td>
<td>2</td>
</tr>
<tr>
<td>6.00000</td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.00255</td>
<td>0.00255</td>
<td>-0.00007</td>
<td>-0.00007</td>
<td>2</td>
</tr>
<tr>
<td>7.00000</td>
<td>-0.00001</td>
<td>-0.00001</td>
<td>0.00094</td>
<td>0.00094</td>
<td>-0.00003</td>
<td>-0.00003</td>
<td>1</td>
</tr>
<tr>
<td>8.00000</td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.00035</td>
<td>0.00035</td>
<td>-0.00001</td>
<td>-0.00001</td>
<td>1</td>
</tr>
<tr>
<td>9.00000</td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.00013</td>
<td>0.00013</td>
<td>-0.00001</td>
<td>-0.00001</td>
<td>1</td>
</tr>
<tr>
<td>10.00000</td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.00005</td>
<td>0.00005</td>
<td>0.00000</td>
<td>0.00000</td>
<td>1</td>
</tr>
</tbody>
</table>

We can note the following points about this example:

- Only 1000 implicit Euler steps were used. This contrasts with the $5 \times 10^6$ steps estimated for the explicit Euler method in Section 1.7; thus there was a reduction of $1/5000$ in the number of steps required by an explicit integrator, which clearly shows the advantage of using an implicit integrator for stiff ODEs. Again, as discussed in Section 1.7, if this
conclusion is not convincing, using \( a = 500,000,000.5, b = 499,999,999.5 \)
would result in a reduction of \( 1/5 \times 10^6 \) steps!

- The two Newton corrections, \( \delta y_1 \) and \( \delta y_2 \), met the tolerance \( \epsilon_p s = 0.00001 \)
with no more than two iterations, which is an indication of the quadratic convergence of Newton’s method, i.e., when Newton’s method works, it
generally is very efficient.

- Accuracy (not stability) was limited by the step size \( h = 0.01 \); this sug-
gests a higher-order BDF method could be used to good advantage (to
increase the accuracy, while maintaining stability). Specifically, the BDF
methods are stable along the entire negative real axis, and therefore
would be stable for the \( 2 \times 2 \) linear problem of Equations C.9 (since the
two eigenvalues are real and negative, e.g., \( \lambda_1 = -(a - b) = -1, \lambda_2 =
-(a + b) = -1,000,000 \)). An extension of Program C.1 for BDF methods
of order 2 and 3 is available from the authors (W.E.S.).

To investigate this last point (the possible advantage of using higher-order
BDF methods), we now consider the use of a stiff ODE integrator in MATLAB,
which varies the order of the BDF method.

### C.3 MATLAB Program for the Solution of the 2x2 ODE System
Using ode23s and ode15s

A MATLAB program that calls \textit{ode23s} and \textit{ode15s} for the solution of the 2x2
ODE system of Equations C.9 is listed below:

```matlab
% 2 x 2 ODE system by variable order BDF

% Global variables
global a b ncall;

% Model parameters
a=500000.5;
b=499999.5;

% Select method
for mf=1:2
% Error tolerances
reltol=1.0e-02;
abstol=1.0e-02;
```
% Cases for changes in error tolerances
for ncase=1:2
    reltol=1.0e-02*reltol;
    abstol=1.0e-02*abstol;
%
% Variables for ODE integration
    t0=0.0;
    tf=10.0;
    tout=[t0:1.0:tf]';
    nout=11;
%
% Initialize number of calls to derivative subroutine
    ncall=0;
%
% Initial condition
    y10=0.0;
    y20=2.0;
    y0=[y10 y20]';
%
% Call ODE integrator
    options=odeset('RelTol',reltol,'AbsTol',abstol);
    if(mf==1)[t,y]=ode23s('ode2x2',tout,y0,options); end
    if(mf==2)[t,y]=ode15s('ode2x2',tout,y0,options); end
%
% Display solution and error
    fprintf('

 mf = %1d
 case = %1d
 reltol = %6.2e
 abstol = %6.2e

','mf',ncase,reltol,abstol);

    fprintf(' t y1e y1 erry1
 y2e y2 erry2
');
    for i=1:nout
        lambda1=-(a-b);
        lambda2=-(a+b);
        exp1=exp(lambda1*t(i));
        exp2=exp(lambda2*t(i));
        y1e=(y10+y20)/2.0*exp1-(y20-y10)/2.0*exp2;
        y2e=(y10+y20)/2.0*exp1+(y20-y10)/2.0*exp2;
        erry1=y1e-y(i,1);
        erry2=y2e-y(i,2);
        fprintf('%5.1f%9.4f%9.4f%15.10f

 %9.4f%9.4f%15.10f
 ','%s','t(i),y1e,y(i,1),erry1,y2e,y(i,2),erry2);
    end

% Next case
fprintf(' ncall = %5d
',ncall);
end
%
% Next method
end
%
% Plot last solution
plot(t,y);
xlabel('t')
ylabel('y_1(t),y_2(t)')
title(' Appendix C, 2 x 2 Linear System')
gtext('y_1(t)');
gtext('y_2(t)');
print appc.ps

Program C.2
Solution of Equations C.9 by $ode_{23}s$ and $ode_{15}s$

We can note the following points about Program C.2:

- The beginning of the program is similar to that of Program C.1; three variables are declared global so that they can be shared with other routines:

  %
  % 2 x 2 ODE system by variable order BDF
%
  % Global variables
  global a b ncall;
  %
  % Model parameters
  a=500000.5;
  b=499999.5;

- An outer loop executes two times; for the first ($mf = 1$), $ode_{23}s$ is called, while for the second, $ode_{15}s$ is called:

  %
  % Select method
  for mf=1:2
    % Error tolerances
    reltol=1.0e-02;
    abstol=1.0e-02;

For each of the two passes through this for loop, the relative and absolute error tolerances for the two ODE integrators are set to $10^{-2}$.
• In a subordinate loop, the error tolerances are reduced by $10^{-2}$ (so that for the two passes through this loop, the error tolerances are set to $10^{-4}$ and $10^{-6}$); in this way, the performance of the integrators can be assessed using the exact solution of the 2x2 ODE problem, Equation 1.17:

```matlab
% Cases for changes in error tolerances
for ncase=1:2
    reltol=1.0e-02*reltol;
    abstol=1.0e-02*abstol;
end
```

• The variables controlling the integration are set:

```matlab
% Variables for ODE integration
t0=0.0;
tf=10.0;
tout=[t0:1.0:tf]';
nout=11;
```

Note that the solution is computed to a final time $tf = 10$ with 11 outputs at an interval of 1.

• A counter for the number of calls to the derivative routine (discussed subsequently) is initialized; also, the initial conditions for the two ODEs (Equations C.9) are set (these are a $2 \times 1$ column vector, which is defined as the transpose of a $1 \times 2$ row vector):

```matlab
% Initialize number of calls to derivative
% subroutine
ncall=0;

% Initial condition
y10=0.0;
y20=2.0;
y0=[y10 y20]';
```

• The MATLAB utility `odeset` is called to set the error tolerances for the subsequent calls to `ode23s` and `ode15s`:

```matlab
% Call ODE integrator
options=odeset('RelTol',reltol,'AbsTol',abstol);
if(mf==1)[t,y]=ode23s('ode2x2',t0,y0,options); end
if(mf==2)[t,y]=ode15s('ode2x2',t0,y0,options); end
```

Note that the two integrators call the function `ode2x2` to define the derivatives (RHS functions) of Equations C.9. Also, the coding to call `ode23s` and `ode15s` is straightforward (which facilitates their use).
• The numerical and exact solutions, and their differences, are displayed in the same way as Program C.1:

```matlab
% Display solution and error
fprintf('mf = %d
 case = %d
 reltol = %e
 abstol = %e
...
', mf, ncase, reltol, abstol);
fprintf('t y1e y1 erry1
 y2e y2 erry2
');
for i=1:nout
    lambda1=-(a-b);
    lambda2=-(a+b);
    exp1=exp(lambda1*t(i));
    exp2=exp(lambda2*t(i));
    y1e=(y10+y20)/2.0*exp1-(y20-y10)/2.0*exp2;
    y2e=(y10+y20)/2.0*exp1+(y20-y10)/2.0*exp2;
    erry1=y1e-y(i,1);
    erry2=y2e-y(i,2);
    fprintf('%5.1f%9.4f%9.4f%15.10f
 %9.4f%9.4f%15.10f
',
        t(i),y1e,y(i,1),erry1,y2e,y(i,2),erry2);
end
% Next case
fprintf('ncall = %d
', ncall);
end
% Next method
end
```

Also, the number of calls to the derivative routine `ode2x2` is displayed at the end of the solution. In this way, we can compare the number of derivative evaluations for the two integrators, and the fixed step integrator of Program C.1.

• Plotting is added at the end to display the solutions from `ode15s` (which, graphically, is identical to the solution from `ode23s`):

```matlab
% Plot last solution
plot(t,y);
xlabel('t')
ylabel('y1(t),y2(t)')
title('Appendix C, 2 x 2 Linear System')
```
Function odev2x2 to define the RHS functions of the ODEs (Equations C.9) is listed below:

```matlab
function yt=ode2x2(t,y)

    % Global variables
    global a b ncall;
    % ODEs
    yt(1)=-a*y(1)+b*y(2);
    yt(2)= b*y(1)-a*y(2);
    yt=yt';
    % Increment number of calls to ode2x2
    ncall=ncall+1;
```

Program C.3
Function odev2x2.m called by Program C.2

We can note the following points about odev2x2:

- The global variables are available for use in calculating the derivatives; also, the counter for derivative evaluations is incremented by 1 each time odev2x2 is called.
- The derivative row vector is transposed into a column vector, which is the required format for odev23s and odev15s.

The output from Programs C.2 and C.3 is listed in abbreviated form below (the output for \( t = 2, 3, \ldots, 9 \) is deleted to reduce the output to reasonable length):

```
mf = 1
case = 1
reltol = 1.00e-004
abstol = 1.00e-004

<table>
<thead>
<tr>
<th>t</th>
<th>y1e</th>
<th>y1</th>
<th>erry1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td></td>
<td>2.0000</td>
<td>2.0000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>1.0</td>
<td>0.3679</td>
<td>0.3684</td>
<td>-0.0004764099</td>
</tr>
<tr>
<td></td>
<td>0.3679</td>
<td>0.3684</td>
<td>-0.0004764099</td>
</tr>
</tbody>
</table>
```
ncall = 324

mf = 1
case = 2
reltol = 1.00e-006
abstol = 1.00e-006

<table>
<thead>
<tr>
<th>t</th>
<th>y1e</th>
<th>y1</th>
<th>erry1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>y2e</td>
<td>y2</td>
<td>erry2</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td></td>
<td>2.0000</td>
<td>2.0000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>1.0</td>
<td>0.3679</td>
<td>0.3678</td>
<td>0.0000421012</td>
</tr>
<tr>
<td></td>
<td>0.3679</td>
<td>0.3678</td>
<td>0.0000420695</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10.0</td>
<td>0.0000</td>
<td>0.0000</td>
<td>-0.0000030593</td>
</tr>
<tr>
<td></td>
<td>0.0000</td>
<td>0.0000</td>
<td>-0.0000030594</td>
</tr>
</tbody>
</table>

ncall = 1591

mf = 2
case = 1
reltol = 1.00e-004
abstol = 1.00e-004

<table>
<thead>
<tr>
<th>t</th>
<th>y1e</th>
<th>y1</th>
<th>erry1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>y2e</td>
<td>y2</td>
<td>erry2</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td></td>
<td>2.0000</td>
<td>2.0000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>1.0</td>
<td>0.3679</td>
<td>0.3680</td>
<td>-0.0001104769</td>
</tr>
<tr>
<td></td>
<td>0.3679</td>
<td>0.3680</td>
<td>-0.0001104769</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Copyright 2004 by Chapman & Hall/CRC
ncall = 139

t y1e y1 erry1
  y2e y2 erry2
0.0 0.0000 0.0000 0.0000000000
  2.0000 2.0000 0.0000000000
1.0 0.3679 0.3679 0.0000002033
  0.3679 0.3679 0.0000002033
. . .
. . .
10.0 0.0000 0.0000 -0.0000002999
  0.0000 0.0000 -0.0000002999
ncall = 233

We can note the following points about this output:

- `ode23s` did not meet the error tolerances, even with 1591 calls to `ode2x2`, e.g., 1.00e—004 vs. −0.0004764099:

  mf = 1
case = 1
reltol = 1.00e-004
abstol = 1.00e-004

t y1e y1 erry1
  y2e y2 erry2
1.0 0.3679 0.3684 -0.0004764099
  0.3679 0.3684 -0.0004764099
ncall = 324

mf = 1
case = 2
reltol = 1.00e-006  
abstol = 1.00e-006  

<table>
<thead>
<tr>
<th>t</th>
<th>y1e</th>
<th>y1</th>
<th>erry1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.0</td>
<td>0.3679</td>
<td>0.3678</td>
<td>0.0000421012</td>
</tr>
<tr>
<td>0.3679</td>
<td>0.3678</td>
<td>0.0000420695</td>
<td></td>
</tr>
</tbody>
</table>

ncall = 1591

However, we should keep in mind that Program C.1 did not produce solutions that were any better than about 2+ figures, e.g., $-0.00183$, with 1000 derivative evaluations (due to the use of a fixed step, first-order BDF):

<table>
<thead>
<tr>
<th>t</th>
<th>dy(1)</th>
<th>dy(2)</th>
<th>y(1)</th>
<th>y(2)</th>
<th>erry(1)</th>
<th>erry(2)</th>
<th>iter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00000</td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.36971</td>
<td>0.36971</td>
<td>-0.00183</td>
<td>-0.00183</td>
<td>2</td>
</tr>
</tbody>
</table>

• *ode15s* did come close to meeting, or exceeded, the error tolerances, e.g., $1.00e - 004$ vs. $-0.0001104769$:

mf = 2  
case = 1  
reltol = 1.00e-004  
abstol = 1.00e-004

<table>
<thead>
<tr>
<th>t</th>
<th>y1e</th>
<th>y1</th>
<th>erry1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.0</td>
<td>0.3679</td>
<td>0.3680</td>
<td>-0.0001104769</td>
</tr>
<tr>
<td>0.3679</td>
<td>0.3680</td>
<td>-0.0001104769</td>
<td></td>
</tr>
</tbody>
</table>

ncall = 139

mf = 2  
case = 2  
reltol = 1.00e-006  
abstol = 1.00e-006

<table>
<thead>
<tr>
<th>t</th>
<th>y1e</th>
<th>y1</th>
<th>erry1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.0</td>
<td>0.3679</td>
<td>0.3679</td>
<td>0.0000002033</td>
</tr>
<tr>
<td>0.3679</td>
<td>0.3679</td>
<td>0.0000002033</td>
<td></td>
</tr>
</tbody>
</table>

ncall = 233
The number of derivative evaluations by \textit{ode}15s was substantially lower than for \textit{ode}23s, which infers better performance of the higher-order methods in \textit{ode}15s. For \textit{ode}23s:

mf = 1
case = 1
reltol = 1.00e-004
abstol = 1.00e-004
ncall = 324

mf = 1
case = 2
reltol = 1.00e-006
abstol = 1.00e-006
ncall = 1591

and for \textit{ode}15s:

mf = 2
case = 1
reltol = 1.00e-004
abstol = 1.00e-004
ncall = 139

mf = 2
case = 2
reltol = 1.00e-006
abstol = 1.00e-006
ncall = 233

\textit{ode}23s and \textit{ode}15s vary the order of the BDF method (in accordance with the table of coefficients in Section 1.7) as well as the integration step in attempting to meet the specified error tolerance. Thus, they are variable step–variable order implementations of the BDF method; i.e., they perform \textit{h refinement} and \textit{p refinement} simultaneously.

To conclude this appendix, we have observed the effectiveness (superior efficiency) of implicit methods, such as BDF, for stiff ODE problems. This improved performance, however, involves greater computational complexity (generally the solution of linear or nonlinear algebraic or transcendental equations (Equations C.2), depending on whether the ODE system is linear...
or nonlinear); therefore implicit methods should be used only if the ODE system is stiff. Also, we should not conclude that a nonlinear ODE system is necessarily stiff, and therefore an implicit integrator is required.

This final discussion suggests a fundamental question: “How do we know if an ODE system is stiff and therefore an implicit integrator should be used?” In the case of linear ODE systems, we can look at the spread in the eigenvalues, as we did for Equations C.9. However, in the case of nonlinear ODEs, eigenvalues are not defined (and therefore cannot be studied for possible stiffness). For this more general case (of nonlinear ODEs), we suggest the following criterion for determining if an implicit integrator should be used:

$$\text{maximum stable step} \ll \text{problem timescale}$$

In other words, observe if the ODE problem timescale is much greater than the largest integration step ($h$) that can be taken while still maintaining a stable solution, i.e., this suggests that stability is the limiting condition on $h$, and therefore the ODE system is effectively stiff so that an implicit integrator should be used.

To illustrate the application of this criterion, for the 2x2 ODE system of Equations C.9 with $a = 500000.5$, $b = 499999.5$, and $\lambda_1 = -(a-b) = -1$, $\lambda_2 = -(a+b) = -1,000,000$, we found in Section 1.7 that (1) the maximum step for a stable explicit solution is $2/1,000,000$, and (2) the timescale for the ODE system is 10 (so that $e^{-110}$ has decayed to insignificance). Thus, application of the preceding criterion gives

$$\frac{2}{1,000,000} \ll 10$$

which implies that an implicit integrator should be used.

To confirm the preceding analysis, Programs c.2 and c.3 were executed with the two MATLAB explicit integrators, ode23 and ode45. This was easily accomplished by changing the following lines:

```matlab
% Call ODE integrator
options=odeset('RelTol',reltol,'AbsTol',abstol);
if(mf==1) [t,y]=ode23s('ode2x2',tout,y0,options); end
if(mf==2) [t,y]=ode15s('ode2x2',tout,y0,options); end
```

to

```matlab
% Call ODE integrator
options=odeset('RelTol',reltol,'AbsTol',abstol);
if(mf==1) [t,y]=ode23('ode2x2',tout,y0,options); end
if(mf==2) [t,y]=ode45('ode2x2',tout,y0,options); end
```

The resulting change in the numbers of calls to the ODE routine ode2x2 is summarized below:
Integrator | Tolerance | Calls to ode2x2
--- | --- | ---
ode23s | $10^{-4}$ | 324
ode23 | $10^{-4}$ | 11, 939, 197
ode23s | $10^{-6}$ | 1591
ode23 | $10^{-6}$ | 11, 939, 440
ode15s | $10^{-4}$ | 139
ode45 | $10^{-4}$ | 19, 283, 629
ode15s | $10^{-6}$ | 233
ode45 | $10^{-6}$ | 19, 283, 713

Clearly, the MATLAB stiff integrators, and even the basic BDF integrator of Program C.1, are substantially more efficient than the MATLAB explicit (nonstiff) integrators (and the same conclusion would be true for the explicit integrators discussed in Chapters 1 and 2).

However, we now have two additional questions to answer in applying the preceding criterion for stiffness (involving the maximum integration step and problem timescale):

1. How do we determine the maximum integration step for an explicit integrator that still produces a stable solution? Answer: In general, by trial and error using a computer program with an explicit integrator. Or, if the computer run times for a stable explicit solution are large, or an excessive number of derivative evaluations is required to maintain a stable solution, an implicit integrator may possibly be used to good advantage.

2. How do we determine the timescale for the ODE problem? Answer: Either from some knowledge of the characteristics of the problem such as physical reasoning, or again, by trial and error to observe when a complete solution appears to have been computed.

In other words, some trial and error with an explicit integrator is generally required. If the computational effort required to compute a complete solution appears to be excessive, switch to an implicit integrator.

Admittedly, this procedure is rather vague (a general, easily applied mathematical test is not available, especially for nonlinear problems), and some trial and error with explicit integrators first, followed possibly by a switch to implicit integrators, may be required (this is the procedure we generally follow for a new ODE problem). The preceding discussion (in this appendix and Chapters 1 to 5) indicates that a spectrum of ODE/PDE problems can be handled with explicit integrators (e.g., the 1x1 and 2x2 ODE problems, and the linear and nonlinear PDE problems), and that only under the condition of stiffness (or stability constraints) is an implicit integrator required. Thus, some judgment based on direct computational experience is required.
Appendix D

Alternative Forms of ODEs

The ODE systems considered previously (in Chapters 1 to 5 and Appendices A to C) all defined the derivatives (RHS of the ODEs) explicitly; that is, only one derivative appeared in each ODE; Equations 1.6 for the 2x2 linear, constant coefficient ODE system are an example (renumbered here as Equations D.1)

\[
\frac{dy_1}{dt} = a_{11}y_1 + a_{12}y_2 \quad y_1(0) = y_{10} \\
\frac{dy_2}{dt} = a_{21}y_1 + a_{22}y_2 \quad y_2(0) = y_{20}
\]  

(D.1)

Equations D.1 are an example of explicit ODEs (not to be confused with explicit ODE integration algorithms); this designation comes from the characteristic that each ODE defines one derivative explicitly.

However, we could consider ODEs in which each ODE contains more than one derivative, e.g.,

\[
c_{11} \frac{dy_1}{dt} + c_{12} \frac{dy_2}{dt} = -ay_1 + by_2 \\
c_{21} \frac{dy_1}{dt} + c_{22} \frac{dy_2}{dt} = by_1 - ay_2
\]  

(D.2)

or in matrix form

\[
\begin{bmatrix}
c_{11} & c_{12} \\
c_{21} & c_{22}
\end{bmatrix}
\begin{bmatrix}
\frac{dy_1}{dt} \\
\frac{dy_2}{dt}
\end{bmatrix}
= \begin{bmatrix}
-a & b \\
b & -a
\end{bmatrix}
\begin{bmatrix}
y_1 \\
y_2
\end{bmatrix}
\]

\[
\begin{bmatrix}
y_1(0) \\
y_2(0)
\end{bmatrix}
= \begin{bmatrix}
y_{10} \\
y_{20}
\end{bmatrix}
\]
For the special case $c_{11} = 1, c_{12} = 0, c_{21} = 0, c_{22} = 1$, is the explicit ODE system (not to be confused with an explicit integrator):

\[
\begin{align*}
\frac{dy_1}{dt} &= -ay_1 + by_2 \\
\frac{dy_2}{dt} &= by_1 - ay_2
\end{align*}
\] (D.3)

which can be written in matrix form (for the general $n \times n$ case) as

\[
I \frac{dy}{dt} = Ay
\] (D.4)

where

- $I$ = identity matrix
- $A$ = ODE coefficient matrix
- $y$ = dependent variable vector

Equations D.2 are an example of a linearly coupled implicit ODE system, with the coupling matrix $M$:

\[
M \frac{dy}{dt} = Ay
\] (D.5)

The term *linearly coupled* comes from the linear coupling (or linear combinations) of the derivatives on the LHS of Equations D.2.

This form of coupled ODEs is common in applications, and therefore library integrators are available to handle such systems. For example, the MATLAB ODE integrators can accept a coupling matrix that is not the identity matrix; \textit{LSODI,1,4 DASSL,2,3,4 RADAU,5,6} and \textit{MEBDFDAE} can also accommodate such coupled ODE systems.

One approach to the solution of Equations D.2 would be to consider them as linear algebraic equations in the derivatives $dy_1/dt$ and $dy_2/dt$. Then these equations can be solved in the usual ways for linear algebraic equations to arrive at the derivatives explicitly. For example, eliminating $dy_1/dt$ from Equations D.2 gives

\[
\frac{dy_2}{dt} = \frac{c_{21} a y_1 - c_{21} b y_2 + c_{11} b y_1 - c_{11} a y_2}{c_{11} c_{22} - c_{21} c_{12}}
\] (D.6)

Higher-order ($n \times n$) linearly coupled ODE systems can be uncoupled by Gaussian elimination or any other established method for simultaneous algebraic equations. Of course, this presupposes that the coupling matrix $M$ is not singular or ill-conditioned (e.g., that $c_{11} c_{22} - c_{21} c_{12} \neq 0$ in Equation D.6). If this is the case, which is common in applications, more sophisticated methods must be applied to perform the numerical integration of the equations, as subsequently discussed briefly.
For example, if in Equations D.2 $c_{11} = c_{12} = 1$, $c_{21} = c_{22} = 0$, the coupling matrix

$$
\begin{bmatrix}
    c_{11} & c_{12} \\
    c_{21} & c_{22}
\end{bmatrix}
$$

is singular. Note that the second ODE is actually an algebraic equation, i.e.,

$$
\begin{align*}
\frac{dy_1}{dt} + \frac{dy_2}{dt} &= -ay_1 + by_2 \\
0 \frac{dy_1}{dt} + 0 \frac{dy_2}{dt} &= by_1 - ay_2
\end{align*}
$$

Thus, Equations D.8 are actually an example of a differential algebraic or DAE system.

Equation D.5 can be generalized to

$$
M(y) \frac{dy}{dt} = Ay
$$

where now the coupling matrix $M(y)$ is a function of $y$; thus, Equation D.9 defines a nonlinearly coupled implicit ODE system.

Finally, if the ODE system is of the form

$$
f(y, \frac{dy}{dt}, t) = 0
$$

Equation D.10 is a fully implicit ODE system; it is also frequently designated as a DAE system since some of the equations defined by $f$ can be algebraic.

All of the preceding ODEs are a special case of Equation D.10 (depending on the form of $f$ in Equation D.10). Library integrators are available, in principle, for all of the preceding forms of ODE systems. In particular, DASSL and RADAU will accommodate Equations D.10 for certain cases of coupling between the ODEs and algebraic equations. The solution of Equation D.10 for completely general forms of $f$ is still an open and active area of research.

In summary, we list these alternate forms of ODEs (beyond the explicit ODEs illustrated by Equations D.4 to indicate that (1) all of these forms occur in applications in science and engineering, and (2) library solvers are available for most of these forms (but without a guarantee for the successful calculation of an accurate solution, especially for Equation D.10, depending on the form of $f$).

A detailed discussion of these alternate ODE forms and the available solvers is beyond the scope of this book. However, the following mathematical software libraries are a good starting point and source of solvers: netlib, gams, and mebdfdae.
References

5. Library ODE/DAE integrators are available from http://hilbert.dm.uniba.it/~testset/software.htm
Appendix E

Spatial p Refinement

In Section 4.1 and Section 5.1 we considered the numerical integration of a PDE (Equation 4.1) in which the finite difference approximation of the second-order spatial derivative $\partial^2u/\partial x^2$ was programed in derv. Since first- and second-order derivatives in space, with their associated boundary conditions, are so commonplace in applications, the calculation of these derivatives can be facilitated by using library routines. We consider here a few library routines for this purpose.

For example, a derv is listed below that can be used in place of the derv in Section 4.1 (Program 4.1.3):

```matlab
function [ut]=derv(n,t,u)

% Function derv computes the derivative vector of the linear PDE problem
% Declare global variables
global nsteps ndss;

% Problem parameters
xl=0.0;
xu=1.0;

% BC at x = 0
u(1)=0.0;

% BC at x = 1
u(n)=0.0;
```

Copyright © 2004 by Chapman & Hall/CRC
\% \texttt{ux}
if \texttt{ndss} == 2 \ [\texttt{ux}]=\texttt{dss002}(x_l,x_u,n,u); \texttt{end}
if \texttt{ndss} == 4 \ [\texttt{ux}]=\texttt{dss004}(x_l,x_u,n,u); \texttt{end}
\%
\% \texttt{uxx}
if \texttt{ndss} == 2 \ [\texttt{uxx}]=\texttt{dss002}(x_l,x_u,n,\texttt{ux}); \texttt{end}
if \texttt{ndss} == 4 \ [\texttt{uxx}]=\texttt{dss004}(x_l,x_u,n,\texttt{ux}); \texttt{end}
if \texttt{ndss} == 42
\texttt{nl}=1;
\texttt{nu}=1;
\texttt{ux}=[0.0;]
[\texttt{uxx}]=\texttt{dss042}(x_l,x_u,n,\texttt{ux},\texttt{nl},\texttt{nu});
\texttt{end}
if \texttt{ndss} == 44
\texttt{nl}=1;
\texttt{nu}=1;
\texttt{ux}=[0.0;]
[\texttt{uxx}]=\texttt{dss044}(x_l,x_u,n,\texttt{ux},\texttt{nl},\texttt{nu});
\texttt{end}
\%
\% \texttt{pdelin}
\texttt{for} \ i=1:n
\texttt{ut}(i)=\texttt{uxx}(i);
\texttt{end}

\textbf{Program E.1}
\texttt{derv} for the solution of Equations 4.1 to 4.4

We can note the following points about \texttt{derv}:

- After setting the boundary values in \(x\), \(x_l = 0.0\), \(x_u = 1.0\), boundary
conditions (Equations 4.3 and 4.4) are programmed:

\%
\% BC at \(x = 0\)
\texttt{u}(1)=0.0;
\%
\% BC at \(x = 1\)
\texttt{u}(n)=0.0;

Note that here we have zeroed the dependent variables, \texttt{u}(1) and \texttt{u}(n),
rather than the time derivatives, \texttt{ut}(1) and \texttt{ut}(n) as in Program 4.1.3. This
difference is due to the way the PDE, Equation 4.1, is programmed at
the end of \texttt{derv}, as explained below.

- The first derivative \(\partial u/\partial x\) is computed by one of two spatial differention
(DSS) routines, \texttt{dss002} or \texttt{dss004}:
% ux
    if ndss == 2  [ux]=dss002(xl,xu,n,u); end
    if ndss == 4  [ux]=dss004(xl,xu,n,u); end

*dss002* implements three point ($O(\Delta x^2)$) finite difference approximations for $\partial u/\partial x$ whereas *dss004* implements five point ($O(\Delta x^4)$) approximations for this derivative. The vector of dependent variables to be differentiated, $u$, is an input to *dss002* and *dss004*, and $\partial u/\partial x$ is returned in the vector $ux$. The choice of the differentiator is through $ndss$, which is set in *intpar* ($ndss$ is added as another parameter to *intpar* of Program 4.1.1, with $ndss = 2$ to call *dss002* and $ndss = 4$ to call *dss004*).

- The second derivative, $\partial^2 u/\partial x^2$, is then computed by differentiating the first derivative $\partial u/\partial x$ (i.e., by *stagewise differentiation*):

  %
  % uxx
  if ndss == 2  [uxx]=dss002(xl,xu,n,ux); end
  if ndss == 4  [uxx]=dss004(xl,xu,n,ux); end

  The second derivative is returned in vector $uxx$.

- For $ndss = 42$ or $ndss = 44$, two differentiation routines, *dss042* and *dss044*, are called that calculate the second derivative, $\partial^2 u/\partial x^2$, directly:

  if ndss == 42
    nl=1;
    nu=1;
    ux=zeros(n,1);
    [uxx]=dss042(xl,xu,n,u,ux,nl,nu);
  end
  if ndss == 44
    nl=1;
    nu=1;
    ux=zeros(n,1);
    [uxx]=dss044(xl,xu,n,u,ux,nl,nu);
  end

$nl = 1$ and $nu = 1$ specify that Dirichlet boundary conditions are used (according to Equations 4.3 and 4.4); if $nl = 2$ and/or $nu = 2$, Neumann boundary conditions are used. For the latter, the first derivative, $ux$, is required and it is therefore an input to *dss042* and *dss044*. In the present case (with Dirichlet boundary conditions), the first derivative is not required and it is therefore zeroed.
The finite difference approximations in \textit{dss042} and \textit{dss044} are \(O(\Delta x^2)\) and \(O(\Delta x^4)\), respectively. For example, for \textit{dss042}, the calculation of the second derivative \(\partial^2 u/\partial x^2\) is done with the following code (taken from \textit{dss042}):

```matlab
%... Grid spacing
dx=(xu-xl)/(n-1);
%... Calculate uxx at the left boundary, without ux
if nl==1
    uxx(1)=((2.)*u(1)
    +(-5.)*u(2)
    +(4.)*u(3)
    +(1.)*u(4))/(dx^2);
%... Calculate uxx at the left boundary, including ux
elseif nl==2
    uxx(1)=((-7.)*u(1)
    +(8.)*u(2)
    +(-1.)*u(3))/(2.*dx^2)
    +(6.)*ux(1)/(2.*dx);
end
%... Calculate uxx at the right boundary, without ux
if nu==1
    uxx(n)=((2.)*u(n)
    +(-5.)*u(n-1)
    +(4.)*u(n-2)
    +(1.)*u(n-3))/(dx^2);
%... Calculate uxx at the right boundary, including ux
elseif nu==2
    uxx(n)=((-7.)*u(n)
    +(8.)*u(n-1)
    +(-1.)*u(n-2))/(2.*dx^2)
    +(6.)*ux(n)/(2.*dx);
end
%... Calculate uxx at the interior grid points
for i=2:n-1
    uxx(i)=(u(i+1)-2.*u(i)+u(i-1))/dx^2;
end
```

This code involves several finite difference approximations at the boundaries for Dirichlet and Neumann boundary conditions. To keep this
discussion to reasonable length, the details are not discussed (they are available in Reference 1). However, the calculation of the second derivative at the interior points by

```matlab
% ... Calculate uxx at the interior grid points
for i=2:n-1
    uxx(i)=(u(i+1)-2.*u(i)+u(i-1))/dx^2;
end
```

directly parallels the coding in derv of Program 4.1.3:

```matlab
% % Interior points
dx=(xu-xl)/(n-1);
dxs=dx*dx;
for i=2:n-1
    ut(i)=(u(i+1)-2.0*u(i)+u(i-1))/dxs;
end
```

- The corresponding code in dss044 for the second derivative at the interior points is

```matlab
% ... i = 3, 4,..., n-2
for i=3:n-2
    uxx(i)=r12dxs*...  
        ( -1.0*u(i-2)...  
          +16.0*u(i-1)...  
          -30.0*u(i )...  
          +16.0*u(i+1)...  
          -1.0*u(i+2));
```

Note that five points (or values of u) are used to calculate the second derivative, while in dss042, only three points are used. This explains the greater accuracy for dss044 (O(\(\Delta x^4\)) for dss044 and O(\(\Delta x^2\)) for dss042). Again, the details of the finite difference approximations in dss042 and dss044 are given in Reference 1.

- Finally, the PDE, Equation 4.1 is programmed:

```matlab
% % pdelin
for i=1:n
    ut(i)=uxx(i);
end
```
The close correspondence between this coding and the PDE is clear; this is possible through the use of the vector \(u_{xx}\) computed by the preceding differentiation (DSS) routines.

Functions \textit{initial} of Program 4.1.2 and \textit{fprint} of Program 4.1.4 are essentially the same (the value of \(ndss\) is printed in \textit{fprint}).

The output from this combination of functions, plus a main program that is also the same as Program 3.1.1, is discussed below. Since this output is quite voluminous, only a summary with key points is given. For \(ndss = 2, n = 21\), the output is as follows:

\begin{verbatim}

euler2a integrator
ncase = 1  neqn = 21  nsteps = 250  ndss = 2

t  u(num)   u(exact)   diff
 0.00  1.000000  1.000000  0.0000e+000
 0.20  0.141180  0.138911  2.2689e-003
 0.40  0.019932  0.019296  6.3550e-004
 0.60  0.002814  0.002680  1.3350e-004
 0.80  0.000397  0.000372  2.4930e-005
 1.00  0.000056  0.000052  4.3644e-006

rkc4a integrator
ncase = 3  neqn = 21  nsteps = 250  ndss = 2

t  u(num)   u(exact)   diff
 0.00  1.000000  1.000000  0.0000e+000
 0.20  0.141177  0.138911  2.2661e-003
 0.40  0.019931  0.019296  6.3470e-004
 0.60  0.002814  0.002680  1.3634e-004
 0.80  0.000398  0.000372  2.5813e-005
 1.00  0.000056  0.000052  4.5242e-006

Copyright © 2004 by Chapman & Hall/CRC

Copyright 2004 by Chapman & Hall/CRC
\end{verbatim}
<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.0000e+000</td>
</tr>
<tr>
<td>0.20</td>
<td>0.141177</td>
<td>0.138911</td>
<td>2.2661e-003</td>
</tr>
<tr>
<td>0.40</td>
<td>0.019931</td>
<td>0.019296</td>
<td>6.3471e-004</td>
</tr>
<tr>
<td>0.60</td>
<td>0.002814</td>
<td>0.002680</td>
<td>1.3333e-004</td>
</tr>
<tr>
<td>0.80</td>
<td>0.000397</td>
<td>0.000372</td>
<td>2.4898e-005</td>
</tr>
<tr>
<td>1.00</td>
<td>0.000056</td>
<td>0.000052</td>
<td>4.3588e-006</td>
</tr>
</tbody>
</table>

rkf45a integrator
ncase = 5 neqn = 21 nsteps = 250 ndss = 2

<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.0000e+000</td>
</tr>
<tr>
<td>0.20</td>
<td>0.141177</td>
<td>0.138911</td>
<td>2.2661e-003</td>
</tr>
<tr>
<td>0.40</td>
<td>0.019931</td>
<td>0.019296</td>
<td>6.3471e-004</td>
</tr>
<tr>
<td>0.60</td>
<td>0.002814</td>
<td>0.002680</td>
<td>1.3333e-004</td>
</tr>
<tr>
<td>0.80</td>
<td>0.000397</td>
<td>0.000372</td>
<td>2.4898e-005</td>
</tr>
<tr>
<td>1.00</td>
<td>0.000056</td>
<td>0.000052</td>
<td>4.3588e-006</td>
</tr>
</tbody>
</table>

rkf45b integrator
ncase = 6 neqn = 21 nsteps = 250 ndss = 2

<table>
<thead>
<tr>
<th>t</th>
<th>u(num)</th>
<th>u(exact)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.000000</td>
<td>1.000000</td>
<td>0.0000e+000</td>
</tr>
<tr>
<td>0.20</td>
<td>0.141177</td>
<td>0.138911</td>
<td>2.2661e-003</td>
</tr>
<tr>
<td>0.40</td>
<td>0.019931</td>
<td>0.019296</td>
<td>6.3471e-004</td>
</tr>
<tr>
<td>0.60</td>
<td>0.002814</td>
<td>0.002680</td>
<td>1.3333e-004</td>
</tr>
<tr>
<td>0.80</td>
<td>0.000397</td>
<td>0.000372</td>
<td>2.4898e-005</td>
</tr>
<tr>
<td>1.00</td>
<td>0.000056</td>
<td>0.000052</td>
<td>4.3587e-006</td>
</tr>
</tbody>
</table>

For this case (n = 21), the errors are relatively large, e.g., for rkf45a at t = 0.20 the error at x = 0.5 is 2.2661e-003:
rkf45a integrator

ncase = 5  neqn = 21  nsteps = 250  ndss = 2

t  u(num)  u(exact)  diff
0.00  1.000000  1.000000  0.0000e+000
0.20  0.141177  0.138911  2.2661e-003

Thus, we would expect that as more points are added (h refinement) and a higher-order spatial differentiator is used (dss044 in place of dss042), which is a form of p refinement in space, the accuracy of the solution would improve. This improvement is demonstrated in the following summary of results from rKF45a at t = 0.2:

ncase = 5  neqn = 21  nsteps = 250  ndss = 2
0.20  0.141177  0.138911  2.2661e-003

ncase = 5  neqn = 31  nsteps = 250  ndss = 2
0.20  0.139828  0.138911  9.1682e-004

ncase = 5  neqn = 41  nsteps = 250  ndss = 2
0.20  0.139476  0.138911  5.6448e-004

ncase = 5  neqn = 21  nsteps = 500  ndss = 4
0.20  0.138921  0.138911  9.6684e-006

ncase = 5  neqn = 31  nsteps = 500  ndss = 4
0.20  0.138913  0.138911  2.1371e-006

ncase = 5  neqn = 41  nsteps = 500  ndss = 4
0.20  0.138912  0.138911  6.5075e-007

ncase = 5  neqn = 21  nsteps = 500  ndss = 42
0.20  0.139476  0.138911  5.6448e-004

ncase = 5  neqn = 31  nsteps = 500  ndss = 42
0.20  0.139162  0.138911  2.5071e-004

ncase = 5  neqn = 41  nsteps = 1000  ndss = 42
0.20  0.139052  0.138911  1.4099e-004

ncase = 5  neqn = 21  nsteps = 500  ndss = 44
0.20  0.138913  0.138911  1.6930e-006
ncase = 5  neqn = 31  nsteps = 500  ndss = 44
0.20  0.138911  0.138911  3.5667e-007

ncase = 5  neqn = 41  nsteps = 1000  ndss = 44
0.20  0.138911  0.138911  1.1461e-007

We can note the following points about this output:

- The number of steps in $t$ to maintain stability changed through the values 250, 500, 1000. Generally this number had to be increased with increasing numbers of grid points, $neqn$ (as $h$ refinement was used), and increasing order of the finite difference approximations (as $p$ refinement was used).
- Generally, the order conditions were maintained when changing the number of grid points, $neqn$. For example, with $ndss = 44$ ($dss044$ was called), $O(\Delta x^4)$, and comparing the $neqn = 21$ and $neqn = 41$ solutions,

$$1.6930 \times 10^{-6} (1/2)^4 = 1.0581 \times 10^{-7} \simeq 1.1461 \times 10^{-7}$$

These order conditions will not be maintained exactly because of the additional error introduced by the integration in $t$, i.e., errors occur because of the approximate spatial and time discretizations. Note that $nsteps = 500$ and 1000 in the preceding solutions so that the change in the number of steps in $t$ complicates the comparisons in $x$.
- Also, the order conditions were maintained when changing the order of the approximations, e.g., $dss042$ (with $O(\Delta x^2)$) to $dss044$ (with $O(\Delta x^4)$). For example, for $neqn = 21$,

$$5.6448 \times 10^{-4} (1/20)^4 (1/20)^2 = 1.4112 \times 10^{-6} \simeq 1.6930 \times 10^{-6}$$

This result clearly indicates the advantage of using higher-order approximations, e.g., $O(\Delta x^4)$ rather than $O(\Delta x^2)$.

Finally, in addition to the routines for $\partial u/\partial x$ and $\partial^2 u/\partial x^2$, i.e., three point ($O(\Delta x^2)$) and five point ($O(\Delta x^4)$) approximations in the six languages, routines with seven point ($O(\Delta x^6)$), nine point ($O(\Delta x^8)$) and 11 point ($O(\Delta x^{10})$) approximations in Fortran and MATLAB are available from the authors; the translation of these routines to the other four languages is straightforward.

References

Appendix F

Testing ODE/PDE Codes

The development of a new ODE/PDE application typically requires some trial-and-error analysis until the application is running correctly. The errors that can occur during this development process are generally of two types:

1. **Compiler errors** resulting from incorrect syntax for the particular compiler (language)
2. **Execution errors** after successful compilation of the source code

We cannot give any specific help with compiler errors since they must be corrected by repeated attempts at compiling the source code until the compiler accepts the source code with no reported errors. The success in the elimination of compiler errors is directly tied to the programmer’s experience with the language.

Execution errors can generally occur for two reasons:

1. The ODE/PDE mathematical model has fundamental flaws that eventually cause arithmetic problems. For example, if the model involves division by zero, or if the model equations are unstable so that eventually the calculations cause an overflow (the calculations produce numbers that exceed the largest number the computer can handle), an execution error will eventually be reported.
2. Errors were made in the programming of the model equations; this could be something as simple as a sign error.

Execution errors are generally the more difficult to correct (compiler errors will be rather explicit and can generally be corrected by reading the compiler error messages and making corrections in the source code). Thus, we present here a method for detecting and correcting execution errors when developing an ODE/PDE application.
Here, then, are the steps that should be most relevant to finding execution errors:

- We assume that some form of numerical integration of an ODE system is a fundamental part of the calculations and the associated source code (PDEs will generally be approximated as a system of ODEs by the method of lines (MOL) as discussed in Chapters 4 and 5).
- Next, we assume that the execution errors are most likely the result of errors in programming the RHSs of the ODEs. Thus, we concentrate on the programming of the ODEs to look for execution errors.
- In all the programming of initial value ODEs we have considered previously, a vector of dependent variables is to be computed (starting with the vector defined as initial conditions). Thus, the principal output of the calculations is the vector of dependent variables as a function of the independent variable (which is the solution to a system of ODEs), and we therefore examine the vector of dependent variables in detail to look for results that are obviously in error, e.g., some “bad numbers.”
- However, the vector of dependent variables is generated by numerically integrating the associated vector of derivatives. So we also should examine in detail the vector of derivatives computed from the RHSs of the ODEs. In fact, it is the vector of derivatives that determines the vector of dependent variables (i.e., that determines the solution). In this sense, the derivatives are as interesting as the dependent variables since they define the solution (through the ODEs).
- To organize these ideas, we can output the dependent variable vector at the beginning of the derivative code or routine (since this vector defines the vector of derivatives through the ODEs).
- We can then output the vector of derivatives at the end of the derivative calculations (the end of the code for the ODEs) to look for possible errors in the programming of the ODEs.

To illustrate this procedure for finding the source of execution errors, we consider again the 2x2 linear ODE system of Equations 1.6. Function *initial* now includes initialization of a counter for the number of times the derivative routine is called:

```matlab
function [u0]=initial(n,t)

% Copyright © 2004 by Chapman & Hall/CRC
```
% Initialize counter for calls to the derivative routine (derv)
ncall=0;
%
% Initial condition vector
u0(1)=0;
u0(2)=2;

Program F.1

\textit{initial} with a counter for the derivative evaluations

Note that \textit{ncall} is declared \textit{global} so that it can be passed to other routines, in this case \textit{derv}.

function [ut]=derv(n,t,u)

% Function derv computes the derivative vector
% of the 2 \times 2 ODE problem
%
% Define global variables
  global ncall;
%
% Heading during first call to derv
  if ncall == 0
    fprintf('

 Output from derv

');
    fprintf(' t u(1) u(2)
');
    fprintf(' ut(1) ut(2)\n');
  end
%
% Display dependent variable vector
  fprintf('%10.4f%10.4f%10.4f
',t,u(1),u(2));
%
% Problem parameters
  a=5.5;
b=4.5;
%
% Derivative vector
  ut(1)=-a*u(1)+b*u(2);
  ut(2)= b*u(1)-a*u(2);
%
% Display derivative vector
  fprintf('%20.4f%10.4f\n',ut(1),ut(2));
%
% Increment counter for calls to derv
  ncall=ncall+1;
% Terminate execution after five calls to derv
if ncall==5 dbstop in derv; end

Program F.2

derv with output of the dependent variable vector and derivative vector

We can note the following points about derv:

• The first time derv is executed (with ncall = 0), a heading for the output from derv is displayed:

```matlab
% Heading during first call to derv
if ncall == 0
    fprintf('
Output from derv

');
    fprintf(' t u(1) u(2)
');
    fprintf(' ut(1) ut(2)

');
end
```

• Since derv has the central function of using the dependent variable vector, in this case \([u(1)\ u(2)]^T\), to compute the derivative vector \([ut(1)\ ut(2)]^T\), two output (fprintf) statements are used:
  
  — The dependent variable vector is displayed at the beginning of derv so that the state of the dependent variables coming into derv can be judged. If they do not look reasonable, then there is probably an error in the numerical integration; e.g., perhaps some of the dependent variables are moving in the wrong direction (due possibly to an error in the programming of the derivatives that follows in derv), or are becoming large due to an instability in the numerical integration (for example, if an explicit ODE integrator is being used and the step size is too large).

```matlab
% Display dependent variable vector
fprintf('%10.4f%10.4f%10.4f
',t,u(1),u(2));
```

  — The derivative vector is then programmed (the central function of derv) using the dependent variables, i.e., the RHSs of the ODEs are programmed. Once all of the derivatives are computed, they are displayed by an output statement (generally close to the end of the derivative routine)

```matlab
% Derivative vector
ut(1)=-a*u(1)+b*u(2);
ul(2)= b*u(1)-a*u(2);
```
The purpose of this output statement for the derivatives is to check if the derivatives appear reasonable (not of the wrong sign, e.g., a dependent variable that should be decreasing should have a negative derivative or too large because units in the model equations are not correct or the numerical integration is becoming unstable).

• Once the input dependent variable vector and output derivative vector are checked, the number of times the output appears should be limited (because the derivative routine typically can be called hundreds or thousands of times during the computation of a complete numerical solution to an ODE problem). In this case, execution of the program is terminated when `ncall` reaches a value of 5 (five calls to `derv`):

```matlab
% Increment counter for calls to derv
ncall=ncall+1;
%
% Terminate execution after five calls to derv
if ncall==5 dbstop in derv; end
```

Note that the MATLAB command `dbstop` is used when `ncall` = 5 to terminate execution.

The output from `derv` is listed below:

<table>
<thead>
<tr>
<th>t</th>
<th>u(1)</th>
<th>u(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ut(1)</td>
<td>ut(2)</td>
</tr>
<tr>
<td>0.0000</td>
<td>0.0000</td>
<td>2.0000</td>
</tr>
<tr>
<td></td>
<td>9.0000</td>
<td>-11.0000</td>
</tr>
<tr>
<td>0.0100</td>
<td>0.0900</td>
<td>1.8900</td>
</tr>
<tr>
<td></td>
<td>8.0100</td>
<td>-9.9900</td>
</tr>
<tr>
<td>0.0100</td>
<td>0.0850</td>
<td>1.8950</td>
</tr>
<tr>
<td></td>
<td>8.0600</td>
<td>-10.0400</td>
</tr>
<tr>
<td>0.0200</td>
<td>0.1656</td>
<td>1.7946</td>
</tr>
<tr>
<td></td>
<td>7.1649</td>
<td>-9.1251</td>
</tr>
<tr>
<td>0.0200</td>
<td>0.1612</td>
<td>1.7992</td>
</tr>
<tr>
<td></td>
<td>7.2101</td>
<td>-9.1704</td>
</tr>
</tbody>
</table>
Not too surprisingly, the output looks reasonable (this is a small, simple ODE problem). As expected, \( u_1(t) \) has a positive derivative (it increases from the initial condition \( u_1(0) = 0 \)) and \( u_2(t) \) has a negative derivative (it decreases from the initial condition \( u_2(0) = 2 \)). If these signs in the derivatives were not observed, we would know something is wrong, probably in the preceding programming of the derivatives.

Also, note that the initial conditions can be checked (in the \( t = 0 \) output). This is an important check (the solution must start out at the right values). Although this check is obvious in this small problem (there are only two initial conditions, \( u_1(0) = 0, u_2(0) = 2 \)), for larger problems, e.g., hundreds or thousands of ODEs, overlooking even one initial condition or using an incorrect value will most likely guarantee that the numerical solution will be incorrect.

The same is true for the derivative calculations in \( \text{derv} \). If we are integrating \( n \) first-order ODEs, we need to program \( n \) derivatives. If we overlook even one derivative, which is easy to do in a large ODE system, the solution will be incorrect. Thus, at the end of \( \text{derv} \) we must have \( n \) good numerical values for the derivative vector.

Also, intermediate calculations before the final calculation of the derivatives are quite common. For example, we might have to solve a set of nonlinear equations, using the ODE dependent variables as inputs. Once these intermediate variables, such as from the solution of a set of nonlinear equations, are computed in \( \text{derv} \), they can be used in the calculation of the derivatives. Of course, errors can occur in these intermediate calculations, and they can be checked by using additional output statements.

Generally, a complete output of all of the variables used in the calculation of the derivative vector can be included in \( \text{derv} \) to ensure that the calculations are done correctly (according to the model equations). This is particularly true for PDE systems in which spatial (boundary value) derivatives are computed in the method of lines. These spatial derivatives can be displayed from \( \text{derv} \) to check their calculation.

Eventually, after detailed checking of the output from \( \text{derv} \), the derivative vector (that is the input to the ODE integrator) will be correct; the output statements in \( \text{derv} \) and the derivative counter can then be removed to compute a complete solution (or these statements can be “commented out” in case they have to be subsequently reactivated for more checking).

Although the preceding method of checking the initial derivative calculations is usually effective in finding and correcting errors, it will not be effective for the case when the initial calculations appear to be correct, but later, the numerical solution develops an obvious problem; e.g., the dependent variables become excessively large, or the compiler reports a \( \text{NAN} \) (not a number). This may be due to integrator instability, or possibly to an error that grows slowly, but eventually causes a calculational failure. The difficulty in finding the cause of such problems stems from the uncertainty in knowing when it occurs (in order to produce some output from \( \text{derv} \), for example), and why. That is why
some knowledge of how numerical ODEs integrators function is important, such as issues of stability, and error monitoring and step size control. For the latter, a common source of failures is to specify unreasonable error tolerances, as discussed in Chapter 1. But in any case, there is no substitute for thorough testing and scrutiny of the computed output.

Finally, two other points can be considered to assist in the checking of computer codes for ODE/PDE problems:

First, we can consider the question of the units of the timescale, e.g., in an application, when we compute the dependent variables of an ODE system as a function of the independent variable \( t \), is the timescale (units of \( t \)) in microseconds, milliseconds, seconds, hours, days, years, etc.? The answer generally is that the units of the timescale are the same as the units of the derivative vector. In other words, the derivatives will have the units of reciprocal time, and whatever those units are will be the units of the timescale.

Additionally, the time units must be the same for all of the computed derivatives. Thus, we cannot calculate the derivative for one dependent variable with the units of seconds, and another derivative with the units of hours. The simultaneous integration of the two derivatives will lead to an incorrect solution. To avoid this problem, the time units of all of the derivatives should be checked to ensure that they are the same.

The same reasoning can be applied to the units of the dependent variables. For example, if an energy balance produces a time derivative with the units of K/min, integration of this derivative will produce a solution with the temperature in K and the timescale in min.

Second, as a word of advice based on experience, we suggest that the development of a new ODE/PDE application should not necessarily be initiated with all of the details of the mathematical model included. This might seem like an inexplicable approach since we eventually want to solve the model equations with the full and complete details we think are necessary to describe the physical system. However, for a relatively complicated model, putting all the mathematical details into the code at the beginning very often guarantees that the code will not execute correctly. If this occurs, the question then that is often difficult or impossible to answer is “What caused the code to fail?” In other words, there are so many details that could be the source of the calculational failure, the identification of which particular detail(s) are the cause of the failure is difficult (although the testing with intermediate output as described previously can help in identifying the cause of the failure).

As an alternate approach, we suggest that the code be built up a little at a time. For example, very simple ODEs can be coded first which can be numerically integrated and the solution checked. The ODEs might even be coded with constant derivatives (and therefore their solutions should be linear in the independent variable); if the derivatives of some of the dependent variables are set to zero, those dependent variables should remain constant during the testing. Then an ODE can be added while the other dependent variables are computed from derivatives that are constant. The code is then
built by adding more ODEs. As this process continues, if a failure occurs, it must be due to the ODEs or mathematical relationships just added, and these can be examined in detail, or they can be reset to the previous condition for which the code executed and some additional testing can be performed to determine why the last step failed. In this way, mathematical details are added until the entire model is coded. The source of any failure along the way can be identified and corrected, particularly by using detailed output from the derivative routine as discussed previously.

In other words, the development of an ODE/PDE code is an experimental, trial-and-error, evolutionary process. Thus, as much as we might like to proceed directly to the solution of the complete mathematical model with all of the details that we think are relevant and should be included, starting out with a simpler ODE/PDE system that has a solution that can be checked, then extending the system through a series of steps, each of which can be checked before going on to the next step, we think is the best way to arrive at a final working code for the problem system of interest.